PhD Thesis

Robust Botnet Detection Techniques for Mobile and Network Environments

Author:

Basil Alothman

First Supervisor:

Dr.Suleiman Yerima

Second Supervisor:

Professor Helge Janicke

A thesis submitted in partial fulfilment of the requirement for the degree of Doctor of Philosophy PhD.

in the

Faculty of Technology

De montfort University

Leicester, United Kingdom

April 2019

Declaration and List of Publication

I, Basil ALOTHMAN, declare that this thesis titled, 'Robust Botnet Detection Techniques for Mobile and Network Environments' and the work presented in it is my own. I confirm that this work submitted for assessment is my own and is expressed in my own words. Any uses made within it of the works of other authors in any form (e.g. ideas, equations, figures, text, tables, programs) are properly acknowledged at any point of their use. A list of the references employed is included. This thesis is written by me and produced using LaTeX.

Professional Conference/Journal Papers (Published):

1. B. Alothman and P. Rattadilok, "Android botnet detection: An integrated source code mining approach," 2017 12th International Conference for Internet Technology and Secured Transactions (ICITST), Cambridge, 2017, pp. 111-115.

2. B. Alothman and P. Rattadilok, "Towards using transfer learning for Botnet Detection," 2017 12th International Conference for Internet Technology and Secured Transactions (ICITST), Cambridge, 2017, pp. 281-282.

3. B. Alothman, "Similarity-based instance transfer learning for botnet detection.", International Journal of Intelligent Computing Research (IJICR) 9, 880-889 (Mar2018).

4. B. Alothman, "Raw Network Traffic Data Preprocessing and Preparation for Automatic Analysis," International Conference On Cyber Incident Response, Coordination, Containment and Control (Cyber Incident), Glasgow, 2018.

5. B.Alothman, H. Janicke, and S. Yerima,(2018) "Class Balanced Similarity-Based Instance Transfer Learning for Botnet Family Classication," The 21st International Conference on Discovery Science (DS2018), Limassol, Cyprus, 2018 Lecture Notes in Computer Science, Springer International Publishing, Cham. pp. 99-113.

Signed: 

Date: 30 April 2019

Abstract

Cybercrime costs large amounts of money and resources every year. This is because it is usually carried out using different methods and at different scales. The use of botnets is one of the most common successful cybercrime methods. A botnet is a group of devices that are used together to carry out malicious attacks (they are connected via a network). With the widespread usage of handheld devices such as smartphones and tablets, networked devices are no longer limited to personal computers and laptops. Therefore, the size of networks (and therefore botnets) can be large. This means it is not surprising for malicious users to target different types of devices and platforms as cyber-attack victims or use them to launch cyber-attacks. Thus, robust automatic methods of botnet detection on different platforms are required.

This thesis addresses this problem by introducing robust methods for botnet family detection on Android devices as well as by generally analysing network traffic. As for botnet detection on Android, *this thesis proposes an approach to identify botnet Android botnet apps by means of source code mining.* The approach analyses the source code via reverse engineering and data mining techniques for several examples of malicious and non-malicious apps. Two methods are used to build datasets. In the first, text mining is performed on the source code and several datasets are constructed, and in the second, one dataset is created by extracting source code metrics using an open-source tool.

Additionally, this thesis introduces *a novel transfer learning approach for the detection of botnet families by means of network traffic analysis.* This approach is a key contribution to knowledge because it adds insight into how similar instances can exist in datasets that belong to different botnet families and that these instances can be leveraged to enhance model quality (especially for botnet families with small datasets). This novel approach is denoted Similarity Based Instance Transfer, or SBIT. Furthermore, the thesis presents a proposed extended version designed to overcome a weakness in the original algorithm. The extended version is called CB-SBIT (Class Balanced Similarity Based Instance Transfer).

Acknowledgements

Firstly, I would like to thank Allah (God) for giving me this opportunity to complete my PhD thesis.

It is not easy to give enough thanks to people who supported me during my time at De Montfort University. But I would like to express my thanks to several people (in no particular order). I would like to extend my thanks and appreciation to my supervisors Dr Suleiman Yerima and Professor Helge Janicke for their support and guidance during my research.

Also, special thanks always go to all my family members and friends who were patient while I was abroad away from my home country and wished me the best success and encouraged me to work hard and submit my thesis as soon as possible. They have always been with me during the good and difficult times.

Finally, I would like to thank my sponsor (Ministry of Education, Kuwait) and Embassy of Kuwait, Kuwait Cultural Office Supervisors who followed the progress of my work and were always available to support me and answer my questions and concerns.

This Research is Dedicated

To my late paternal grandfather Mr. Nasser Alothman *who taught me the importance and power of relationships, tolerance and giving.*

To my late maternal grandfather Mr. Homoud Almogahwi *who instilled in me the endurance and passion required for hard work. He supported me financially, morally and provided me with books at a young age (he owned a large press company that generously printed and distributed books around GCC countries).*

To my parents Yousef Alothman and Nadia Almogahwi *who were always proud of me and influenced me to be a computer scientist. They never stopped showering me with their kindness, generosity and love.*

To my beloved wife Mariam Aldalali *for her help, empathy and endless love. She is my real constant source of encouragement, strength and support.*

To my children Yousef, Nadia, Loulwah and Bader *in whose eyes I always saw the motivation and hope and for being quiet, obedient, respectful and cooperative with us.*

Contents

List of Figures

List of Tables

Abbreviations

ITU	**I**nternational **T**elecommunication **U**nion
iOS	**i**Phone/iPad/iPod **O**perating **S**ystem
SMTP	**S**imple **M**ail **T**ransfer **P**rotocol
P2P	**P**eer **T**o **P**eer
PCAP	**P**acketCAP **CAP**ture
Bot	Ro**Bot**
Net	**Net**work
Botnet	Ro**Bot** **Net**work
C&C	**C**ommand **and** **C**ontrol
DoS	**D**enial **o**f **S**ervice
DDoS	**D**istributed **D**enial **o**f**S**ervice
IRC	**I**nternet **R**elay **C**hat
HTTP	**H**yper **T**ext **T**ransfer**P**rotocol
SQL	**S**tructured **Q**uery **L**anguage
IP	**I**nternet **P**rotocol
MIL	**M**ultiple **I**nstance **L**earning
ANNs	**A**rticial **N**eural **N**etworks
MLFF	**M**ulti-**L**ayer **F**eed-**F**orward
SSL	**S**ecure **S**ocket **L**ayer
MLP	**M**ulti-**L**ayer **P**erceptron
BBNN	**B**lock-**B**ased **N**eural **N**etwork
FPGA	**F**ield **P**rogrammable **G**ate **A**rray
SVM	**S**upport **V**ector **M**achine
OC-SVM	**O**ne- **C**lass-SVM
KNN	**K-N**earest **N**eighbours

NB	**N**aive **B**ayes
RF	**R**andom**F**orest
OPFC	**O**ptimum-**P**ath **F**orest **C**lustering
SOM	**S**elf-**O**rganising **M**aps
DTs	**D**ecision **T**rees
REPTree	**R**educed **E**rror **P**runing **Tree**
TL	**T**ransfer **L**earning
NoTL	**NO T**ransfer **L**earning
APK	**A**ndroid **P**ackage **K**it
ISCX	**I**nformation **S**ecurity **C**enter of **e**Xcellence
SMS	**S**hort **M**essage **S**ervice
LAH	**L**ist **A**bbreviations **H**ere
JAR	**J**ava **AR**chive
NLP	**N**atural **L**anguage **P**rocessing
API	**A**pplication **P**rogramming **I**nterface
JD	**J**ava **D**ecompiler
GUI	**G**raphical **U**ser **I**nterface
TF-IDF	**T**erm **F**requency **a**nd **I**nverse **D**ocument **F**requency
SCM	**S**ource **C**ode **M**etric
SMO	**S**equential **M**inimal **O**ptimisation
Acc	**Acc**uracy
ERR	**ER**ror **R**ate
FPR	**F**alse **P**ositive **R**ate
FNR	**F**alse **N**egative **R**ate
TPR	**T**rue **P**ositive **R**ate
TNR	**T**ue **N**egative **R**ate
AUC	**A**rea **U**nder the**C**urve
SBIT	**S**imilarity **B**ased **I**instance**T**ransfer **L**earning
CB-SBIT	**C**lass **B**alanced **SBIT**
Sim	**S**imilarity
SMOTE	**S**ynthetic **M**inority **O**ver-sampling **TE**chnique
UK	**U**nited **K**ingdom
CSV	**C**omma- **S**eparated **V**alue

FTP	**F**ile **T**ransfer **P**rotocol
PCA	**P**rincipal **C**omponent **A**nalysis
PLS	**P**artial **L**east **S**quares
LOF	**L**ocal **O**utlier **F**actor

Symbols

$\mathcal{X}$	Feature Space
$\mathbf{X}$	Data
$P(\mathbf{X})$	The marginal probability distribution (for $\mathbf{X}$ in $\mathcal{X}$)
$\mathbf{x}^{(i)}$	The i^{th} instance in $\mathbf{X}$
$\mathcal{D}$	A Domain
$\mathcal{D}_{\mathcal{S}}$	A Source Domain
$\mathcal{D}_{\mathcal{T}}$	A Target Domain
$\mathcal{T}$	A Task
$\mathcal{T}_{\mathcal{S}}$	A Source Task
$\mathcal{T}_{\mathcal{T}}$	A Target Task
$\mathcal{Y}$	Label Space
$\mathbf{y}^{(i)}$	The label of the i^{th} instance in $\mathbf{X}$
$f(\bullet)$	A Predictive Function
$\in$	is in
$\neq$	is not equal to
$P(x)$	Probability of x
$P(x\|c)$	Probability of x given c
$\sum$	Summation
$\sum_{i=1}^{n}$	Summation of all elements from 1 to n
x_i	The ith element in x

Chapter 1

Introduction to the thesis

1.1 Introduction

The Internet is a very busy network that sees huge amounts of data being transferred every day. Users and machines, communicate by sending and receiving various types and amounts of data. This means that despite the usefulness of such communication platform, there are several ways to cause harm. One of these ways is to communicate with someone while pretending to be someone else. Another way is by installing tools onto other people's devices in order to spy on them or to steal some sensitive information. Also, it is possible to use such installed tools to attack others who might be thousand of miles away. These electronic attacks can be launched against personal or corporate computers and networks. One of the existing methods for executing such harmful attacks is via botnets. Botnets are groups of networked devices that can be exploited to carry out malicious attacks.

The dangers of botnets are becoming more widespread; an example of this is the WannaCry attack that caused many major institutions in several countries to struggle to perform their services (Kalita, 2017). The research community has recently been actively trying to develop automatic techniques to identify botnets in order to stop their harmful activities.

Large amounts of money is lost every year due to botnet activities. For example, reports from reliable sources suggest that more than 65 Billion US Dollars were lost in 2005 and the entire damages caused by spam bots were estimated to have a cost of around 100 Billion US Dollars in 2007 alone (International Telecommunication Union, 2017). It is not a secret that organised cyber crimes are becoming widespread as they can be a profitable venture within short periods of time. It is estimated that cyber crime can cost the global economy up to 600 billion US Dollars a year (Healey and Knake, 2018). This is emphasised in a report published by the WhiteHouse where it is estimated that the US economy suffered a loss of more than 108 Billion US Dollars in 2016 (The Council of Economic Advisers, 2018). Hence, researchers have been working actively to develop effective techniques to detect, and protect from, such malicious attacks.

Normally, machines like these are known as bots and such networks are known as *botnets*. More on the definition and architecture of botnets is provided throughout this thesis (especially in Chapters 2 and 3). It is believed that hundreds of millions of computers that are connected to the Internet are infected each year, which results into more than 15 victims each second (Demarest, 2014). These networks represent a serious threat because they can be exploited to carry out malicious and illegal actions which can have high level damages. In fact, existing research indicates that a large number of attacks was because of the Internet of Things (IoT) and its poor security model (Crosbie, 2016). It is estimated that the number of units installed in the IoT will surpass 25 billion by

2020 (Middleton et al., 2013). Hence, the potential number and magnitude of attacks via the IoT is likely to increase.

It is indeed a source of concern to find out that the number of distributed denial of service (DDoS) attacks is on the rise (Wang et al., 2018) and the ways attacks are carried out (in terms of diversity, intensity and duration) is increasing every year (Verisign DDoS Report, 2018).

Android is one of the most popular smartphone operating systems which keeps growing among smart-device users. This operating system is open source, user-friendly and it is relatively easy to write Java applications that run smoothly on it. Its popularity makes it one of the default targets for malicious cyber-attacks. It can be used to launch attacks or it can also be the victim of malicious attacks. Android's play-store is not very restrictive which makes installing malicious apps easy.

It is common for botnet developers to target smartphone users in order to install their malicious tools on a large number of devices. This is often done to gain access to sensitive data such as credit card details, or to cause damage to individual hosts or organisation resources by executing denial of service attacks. With the large number of Android apps being released everyday it is difficult to know whether or not an app is safe to install. To overcome this issue, among other challenges, it would be useful to automate the process of checking how safe a new app is (i.e. to use the smartphone itself to predict whether or not an app is safe).

Because botnets run on networked devices, they normally communicate with each other, and more fundamentally, with a central point known as the Command and Control (C&C for short). The communication takes place to send/receive commands and

responses/results. Botnets use various network protocol to communicate safely and evade any protection and detection systems. For example, many botnets use the Internet Relay Chat (IRC) as well as Simple Mail Transfer Protocol (SMTP) protocols.

Also, botnets can be formed with different architectures. For example, botnets can have the traditional client/server network architecture where several clients connect to the same server (or group of servers). This model is known as the centralised C & C. While this architecture can be easy to develop, it is also easy to combat against. If the communication between the infected machines and the server(s) is terminated, then the botnet is effectively stopped. Another network model that is adapted by some botnets is the Peer-to-Peer configuration (P2P for short). This architecture is more difficult to combat than the centralised approach because of the large number of connections that can involve thousands or millions of devices.

1.2 Motivation

The work presented in this thesis primarily focuses on two problems. The first problem is botnet detection on the Android operating system by means of source code mining and analysis. The approach developed in this thesis analyses the source code of a given android app and attempts to identify whether it is botnet or normal. The second problem is more generic as it focuses on botnet detection and identification via analysis of network traffic. The network based detection is tackled by developing a novel transfer learning algorithm as will be explained in this thesis.

As for botnet detection on Android, the main motivation for the approach presented in this thesis is to develop a proactive method that attempts to identify the danger before it occurs. Android users can easily install apps from both official Google play store and

third party app markets. The problem here is that the only way users can trust an app is via reading comments and reviews written by other users (who may also be malicious actors). The approach presented in this thesis protects users against this by introducing a method for analysing the source code of Android apps before installing them. The method reverse engineers the Android apps and obtains their source code. Then, from the obtained source code, it uses two different approaches to create datasets suitable for machine learning and data mining. This is all done after only downloading the Android app and without installing and running it. It is worth mentioning here that although this thesis focuses on Android apps, the proposed approach can be used in other mobile operating systems. For example, applications running on the iPhone operating system (iOS) can also be reverse engineered (Joorabchi and Mesbah, 2012) and their source code can be analysed using the same method.

Regarding the second problem addressed in this thesis, the main motivation for developing a transfer learning approach for the precise detection of botnet traffic can be summarised by providing an example (an example is provided in Figure 1.1 later in this section). From reviewing existing approaches, it can be noticed that many of them target specific botnets. On the other hand, many approaches try to identify any botnet activity by analysing network traffic. They achieve this by concatenating existing botnet datasets to obtain larger datasets, building predictive models using these datasets and then employing these models to predict whether network traffic is safe or harmful.

The problem with the first approach is that data is usually scarce and costly to obtain. By using small amounts of data, the quality of predictive models will not be optimum. On the other hand, the problem with the second approaches is that it is not always correct to concatenate different datasets (i.e. datasets containing network traffic from

different botnets). Datasets can have different distributions which means they can downgrade the quality and predictive performance of machine learning models.

The proposed ideas in this thesis are based on using transfer learning. In more detail, instead of immediately concatenating datasets that belong to different botnets, this thesis suggests using transfer learning to carefully decide what data to use in concatenating such datasets. The main hypothesis is: *Performance can be improved by using transfer learning techniques across datasets containing network traffic from different botnets. This should be done instead of blindly concatenating datasets.*

So, before providing any further details, one can ask: what is transfer learning?. We as human beings have the ability to utilise past learning experiences when we are faced with new tasks. For example, when someones knows how to ride a bike, can he/she benefit from this experience when they learn how to drive a car?. Our level of mastering the new task depends on how much it is related to our past task. In machine learning, the sub-field that attempts to apply this experience, or knowledge, transfer is known as Transfer Learning. *It is typically employed when there is little, or limited, amounts of labelled data in one task (usually called the target task), and plenty of data in another related task (usually called the source task).* The idea here is that using the target data only can lead to obtaining models with poor performance since there is not sufficient data. By transferring knowledge from the source task(s) the model quality can be improved.

The problem of labeled data scarcity is common in machine learning. In many fields it can be too costly to obtain labeled data. Examples of fields where labeled data can be highly costly to obtain are: data for cancer patients, data for new botnets or viruses and data for undersea studies. One can evaluate existing traditional machine learning

algorithms. Because of the small size of data, the performance of these algorithms can be poor. In order to enhance performance, one idea is to collect more data (which can cost significant amounts of money, time and effort). Fortunately, in many cases, there exists plenty of data in domains that are close (or related) to the domain under study. This is where transfer learning comes into play.

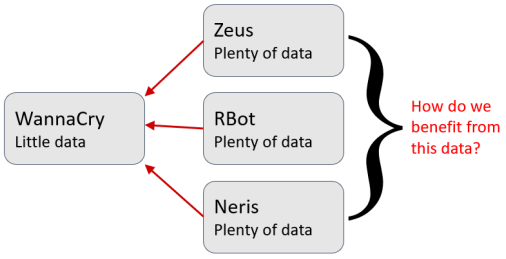


FIGURE 1.1: Example Transfer Learning Scenario

The example diagram in Figure 1.1 illustrates the idea. For WannaCry, the available data is limited and insufficient to create a highly accurate model. On the other hand, there is plenty of data for each of the other three botnets. The idea is to use transfer learning to augment the WannayCry dataset so that more accurate predictive models can be created for WannaCry botnet detection.

1.3 Methodology

The work in this thesis focuses on two main problems as shown in Figure 1.2. The first challenge was to develop a source code mining approach for the analysis of Android apps. The second challenge was to develop a novel approach for the detection of botnets via analysis of network traffic. For both parts, existing literature was reviewed and limitations were identified. Although there is existing work that attempts to address these problems, techniques developed in this thesis add new contributions to the field as will explained throughout the remaining chapters.

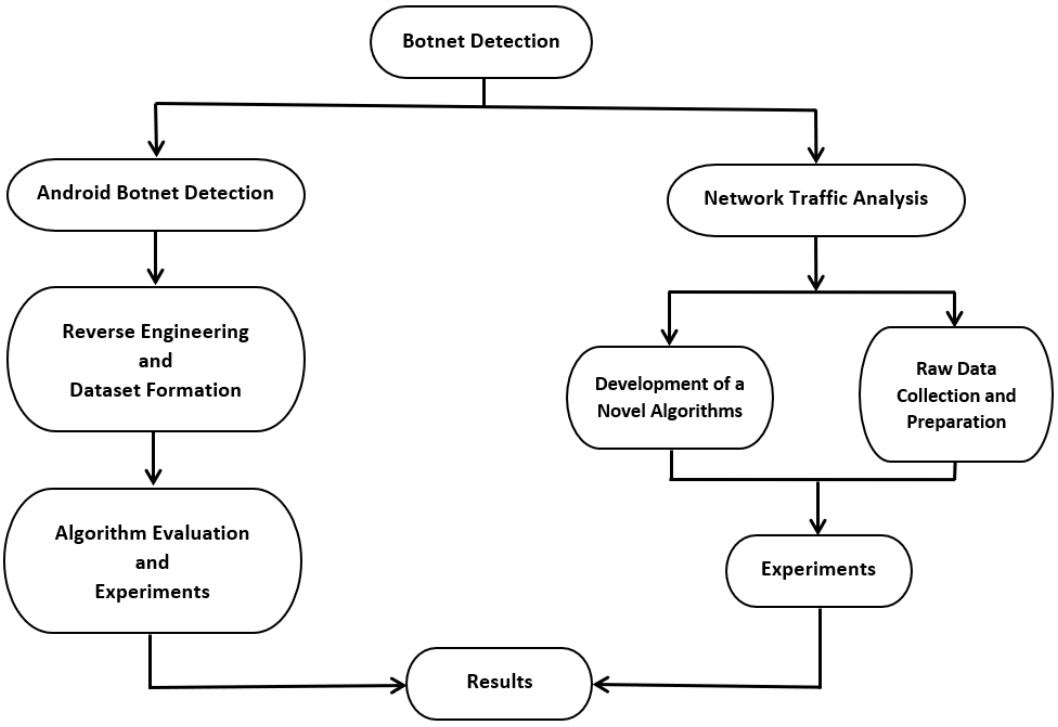


FIGURE 1.2: Thesis Structure

1.3.1 Android Botnet Detection

The Android apps used in this work were collected from existing repositories. Two types of of apps were collected: *normal* and *botnet*. As the main focus is on botnet apps, it was ensured that the normal apps were network apps (i.e. apps that work as part of a network such as messaging and photo sharing apps). This is to make sure the comparison is done using apps that are similar in their underlying architecture. An open source tool was used to reverse engineer these apps and generate their source code. Afterwords, existing text mining approaches were used to form a dataset using the source code. At the same time, a freely available source code analysis tool was used to extract several metrics and for a separate dataset. These datasets were then used to evaluate the performance of several classical machine learning algorithms.

1.3.2 Network Traffic Analysis

The network traffic data used in the network based approach was obtained from an open source repository. The data was in raw format which means it was not immediately suitable for machine learning tools. Therefore, several steps were taken to transform it into a usable format. Subsequently, a novel transfer learning algorithm was developed and implemented. This novel algorithm was then evaluated extensively (using this dataset) and extended to enhance its performance and reduce its limitations.

1.4 Aims and Objectives

Although the previous sections have introduced several key points about the work carried out in this thesis, this section lists the aims and objectives of this thesis:

1.4.1 Aims:

The aims of this thesis are:

1. Develop a robust proactive approach for botnet app detection on android systems
2. Evaluate the proposed approach for botnet app detection on android systems
3. Develop a novel approach for botnet family classification with improved accuracy
4. Evaluate the proposed botnet family classification approach through extensive experiments and comparison with other classification approaches

More details about these points is provided in Section 1.6.

1.4.2 Objectives:

In order to achieve the aims, the following steps are going to be performed to develop the source code mining approach:

1. Collect various botnet and normal android apps and use reverse engineering techniques to obtain their Java source code
2. Use text preprocessing techniques to transform the Java source code into format suitable for machine learning tools
3. Design and implement source code analysis techniques to distinguish between source code of botnet apps from source code of normal apps
4. Run experiments using several classifiers and find out which classifier works best on which dataset

In addition, the following steps are going to be performed to develop the novel transfer learning approach:

1. Collect network traffic data that contains traffic from various botnets as well as normal network traffic
2. Preprocess the data to transform it into format suitable for machine learning tools
3. Design and implement the novel transfer learning algorithm
4. Conduct several experiments to evaluate the performance of this novel algorithm and identify its strong and weak points
5. Extend this novel algorithm to overcome its weaknesses

6. Conduct several experiments to compare the performance of this novel algorithm against the performance of existing commonly-used freely available algorithms

1.5 Hypotheses

The following hypotheses will be tested in the course of this thesis.

Hypothesis 1: Analysis of source code of Android apps can be an effective proactive method for the detection and identification of botnet apps. This thesis proposes an approach to reverse engineer Android apps, obtain their source code and mine this source code to predict whether an app is a bot or not. In other words, this hypothesis states that:

> *Android normal and botnet apps can be distinguished by using machine learning methods to analyse and gain insight into their source code*

Chapter 3 of this thesis presents the development and evaluation of a source code mining approach. The findings of this work show that this simple yet powerful method can indeed produce accurate and reliable results.

Hypothesis 2: Where limited data is available for a target task, data available aplenty for one or more related but different task(s) (known as the source task(s)) can be exploited to enhance learning in the target task. This can be achieved by developing a transfer learning approach that carefully selects data from the source task(s) and transfers it to the target task. In other words, this hypothesis states that:

> *Performance can be improved by using transfer learning techniques across datasets containing network traffic from different botnets.*

A novel approach is developed, extended and compared to other existing approaches in Chapter 4. An extensive evaluation of this approach using different types and sizes of real-world datasets is presented in Chapter 5. The novel approach is shown to outperform existing classical and commonly used approaches.

1.6 Key Contributions

The following points provide a summary of the contributions of this thesis to the field of using Machine Learning for botnet detection:

1. **A Robust Integrated Source Code Mining Approach for Android Botnet Detection:** this approach provides a method that can used to detect botnet android apps by reverse engineering their source code and then analysing it. This method is useful because it helps in detecting botnet apps before users run them. This means that the proposed method attempts to stop any harm before it happens by being proactive. The idea of this methods is based on using text mining techniques to gain insight into the source code of Android apps and classifying them accordingly.

2. **A Novel Similarity Based Instance Transfer Learning approach for Botnet Family Classification:** a novel instance transfer learning approach is developed and evaluated in this work. The main idea of this novel approach is to measure the similarity between instances in different datasets and to transfer highly similar instances to the smaller dataset. The instances are transferred from the Source dataset(s) to the Target dataset. Although this approach is simple and easy to implement, it is powerful and effective (in its accuracy and speed) as will be shown throughout the thesis.

3. **Class Balanced Similarity-Based Instance Transfer Learning for Botnet Family Classification:** this approach extends the Similarity Based Instance Transfer approach mentioned in the previous point. It works in exactly the same way but adds an extra step to ensure class balance in the resulting target dataset. In short, it only keeps instances transferred from source datasets so that the classes in the target dataset have a similar percentage (i.e. a similar proportion in the target dataset). This methods helps in avoiding overfitting and make interpreting models easier.

4. **A Systematic Method for Transforming Raw Network Traffic Data into a format suitable for Machine Learning and Data Mining:** a major part of this thesis is focused on botnet detection via network traffic analysis. However, open source network traffic data usually exists in raw format (known as PCAP format). Therefore, the thesis provides a systematic method that can be applied to not only transform this raw format into a format suitable for machine learning and data mining, but also to gain insight into the data, inspect and visualise it.

1.7 Thesis Overview

Here a summary of the structure of the thesis is provided.

- **Chapter 1:** This chapter presents an introduction to the main research idea, motivation, research scope and limitation, research hypothesis and thesis contributions. The proposed approaches are introduced at a high level; detailed explanations will be provided throughout the remaining chapters of this thesis. In addition, the chapter ended with an overview of the structure of this thesis.

- **Chapter 2:** Chapter two contains the background and literature review. It contains sections on what botnets are, some key concepts that must be understood in order to understand botnets, the anatomy of a botnet attack, botnet topologies and architectures, and several botnet examples. In addition, this chapter provides a summarised introduction to machine learning in general, and transfer learning in particular. It mainly focuses on inductive transfer learning because it is the branch under which this thesis falls. Additionally this chapter has a review of several existing botnet detection techniques that are related to the work in this thesis.

- **Chapter 3:** The third chapter contains a detailed explanation of the integrated source code mining approach that is developed for botnet detection on Android. The chapter starts by providing a summary of existing approaches and then explains in detail how Android apps were reverse engineered to obtain their source code. Next, an explanation of how the source code was used to create two different types of datasets is given. This is followed by an overview of the machine learning methods that were used and a detailed experimental evaluation of their performance.

- **Chapter 4:** Chapter four has the work carried out on network traffic analysis. It explains the novel transfer learning approach that was developed as part of this thesis. Additionally, this chapter defines *similarity*, how it can be calculated and how it was used as part of the novel approach.The chapter also contains an explanation of two existing well known algorithms as they are used in the experimental evaluation. In addition, a major part of this chapter is a section of some of the limitations of the novel approach and an extension to this approach.

- **Chapter 5:** The fifth chapter contains two main sections. One of them is on how to preprocess raw network traffic data, extract useful information from it and transfer it into a format suitable for machine learning tools and platforms. This section provides several steps that can be applied on such data and provides several examples and visualisations. It is then followed by an example use case where these steps were applied to an existing freely available raw data. The second main section of this chapter contains a detailed experimental evaluation using the resulting network traffic dataset as well as some text data. This chapter also includes several performance comparisons such as comparing the novel algorithm (and its extended version) against two common existing algorithms.

- Chapter 6: This chapter concludes the thesis and presents limitations and future work. It is written so that each approach contains a separate section for conclusions, limitations and future work. This is done to keep the explanation focused and to make easier for the reader to understand.

1.8 Summary

In this chapter an introduction to the main research idea was provided to the reader and the hypothesis was stated. The proposed approach was also introduced at a high level; a detailed explanation will be provided throughout the remaining chapters of this thesis. In addition, we have also listed our main contributions and publications. The chapter ended with an overview of the structure of this thesis.

Chapter 2

Background and Literature Review

Before delving into details of the techniques developed in this thesis, it is logical to provide an overview of some essential concepts. It comes as no surprise that there is a large number of techniques for botnet detection and prevention; some of which date back to the 1980s. The contributions of this chapter include an introduction to botnets in order to familiarise the reader with what they are, how they work and the anatomy of their attacks. In addition, the chapter also provides an overview of reverse and re-engineering as the former technique is employed in the work carried out in this thesis. This is followed by a brief overview of the field of machine learning and the sub-field of transfer learning. After that the chapter presents an overview of the most recent botnet detection approaches. A summary of Android botnet detection methods is provided. Also, the overview includes existing botnet detection work that uses transfer learning. As for botnet detection techniques that use traditional machine learning, only the most recent approaches are considered.

2.1 Background

2.1.1 Botnet Introduction

This chapter begins by providing a definition of botnet and then continues with explaining other important concepts and aspects.

2.1.1.1 Botnet Definition

There are several definitions of botnets in the literature. A general definition is: A **botnet** is a group of networked devices that are used to carry out malicious attacks. Examples of such devices are desktop computers, laptops, smartphones and tablets. These devices, known as hosts, are normally under the remote control of another device known as the botmaster (Haddadi et al., 2014).

For malicious users, this configuration is advantageous because the device carrying out attacks, or malicious activities in general, is not theirs. This is because the communication between the botmaster and hosts, or botclients, can be done via Internet Relay Chat (IRC) channels. Using such channels makes it difficult to trace back because attackers can use an obfuscating proxy to send the commands through. Furthermore, attackers can also use tools to send commands via multiple hops to add more complexity (Schiller and Binkley, 2007).

It can be observed that the word botnet consists of two words; *bot* and *net*. A bot is a computer program, a script or an application, that executes tasks automatically. This means that a bot can be useful in cases when automation is required. An example is placing online bids such as on ebay. However, the type this thesis focuses on is the one that is programmed to receive remote commands to perform dangerous actions.

The word net means that several bots (thousands or even millions) are run on networked machines to performs tasks on a large scale. These machines are considered as botclients.

Botnets are considered by many as the core of cyber-crime as they are used to gain control of, and use, a large number of connected devices to send commands to perform harmful tasks such as information theft and industrial espionage (Kirubavathi and Anitha, 2016). In general, they are among the most famous threats that are difficult to mitigate and protect against (Acarali et al., 2016).

One of the most well-known attacks over the internet is the distributed denial of service, or DDoS, attacks. Although botnets are used to carry out such attacks, they are mostly used for spam attacks, or to steal sensitive information such as login credentials and credit card details. Additionally, they are also used to commit fraudulent attacks on banks, bank details and organisations (Kirubavathi and Anitha, 2016).

The ways botnets spread vary. According to (Borgaonkar, 2010) they can be injected into remote machines via using social engineering tricks on chatting applications and hyper-text transfer protocol (HTTP) based communication tools in general. Another way is the use of advanced double fast-flux service networks and structured query language (SQL) injection attacks (Sood et al., 2016).

2.1.1.2 Key Concepts

In order to understand how botnets work, it is important to be familiar with the following concepts which are key in the configuration and functioning of botnets (the following overview was summarised by (Tiirmaa-Klaar et al., 2013)).

- **Network:** the first concept to be familiar with is computer networking. Botnets are spread over a large number of devices that communicate with each other and with their botmaster. This communication facilitates the ability to send/receive commands and updates.

- **Machines are Compromised:** Normally the botclients are compromised devices. What this means is that these clients can be used to perform attacks unwillingly and unknowingly. In other words, these devices are exploited and used to participate in botnets without the knowledge and permission of their real users or owners.

- **Remote Control:** the compromised devices mentioned in the previous point are normally controlled remotely by a botmaster. They receive commands and communicate in a Command and Control configuration (known as the C&C). This allows the exploiter to use some or all of the bots in the botnet as the attack they are trying to perform requires. This control can be in one of several structures (See section 2.1.1.3 for more detail).

- **Remote Controller:** The previous point talked about the compromised devices being remotely controlled. This process is performed by a malicious person (the exploiter). The remote controller is usually the botmaster mentioned in Section 2.1.1.1. This person usually wishes to execute some illegal activities and harmful attacks. An example attack was the DDoS attack that was launched against Estonia (Robinson and Martin, 2017).

2.1.1.3 Anatomy of a Bot Attack

Botnets follow a systematic way to launch attacks. As shown in Figure 2.1, the first step is for the botmaster to infect a victim with a bot. As mentioned previously, there are several ways to infect a victim. Note that the number of infected victims can be large and, therefore, the attacker can have an army of bots under his/her control. After the victims are infected, they connect to the C&C server and wait for instructions. This connection can be established using one of the known protocols such as HTTP or IRC. Then, the C&C server sends its commands to the victims which in turn execute the commands and report back the results to the C&C.

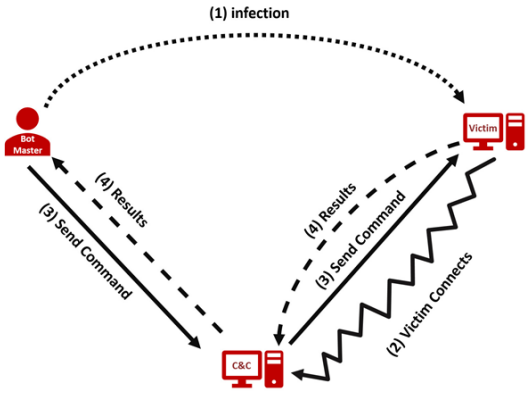


FIGURE 2.1: Anatomy of a Botnet Attack

2.1.1.4 Botnet Topologies

Botnets have their own characteristics and components. They can normally be divided into smaller entities such as botmasters, bot clients, bot servers, bot victims and nodes. The botmaster is the attacker (i.e. controller of the botnet). This is usually the developer of the botnet (i.e. malware management or bot controller). The bot client/host is the target device which the botmaster wants to control and use. The bot Server Command & Control (C&C) is the Command and Control server of the bots which receives commands

from the botmaster through some command and control (C&C) infrastructure to control and give orders to the bot client. In general, there are three types of bot servers C&C: Centralised (one host), Peer2Peer (one to one) and distributed client (random). Nodes are the bots which will be used to attack the victims. Botnet victims, or the botnet customers, receive the planned attack from the attacker. An example bot attack is through sending or flooding the target system with a huge amount of any kind of data to disrupt the system.

Centralised Topology (Star & Distributed cluster)

This configuration is like the usual client-server model where a client connects to a server, and the server sends commands and receives reports/results from the client. In such C&C architecture, all bots connect to the botmaster (see Figure 2.2). It is worth mentioning here that the botmaster itself can consist of more than one device (i.e. it can be a group of machines instead of a single machine as in Figure 2.2). These machines are usually responsible for transmitting commands to bots. This topology offers the advantage of reliable coordination between the bots and their botmaster. Also, it speeds up reaction time and it makes status monitoring easy for the botmaster. On the other hand, the C&C server(s) in this architecture is always a single point of failure. This means that once a botnet is identified, eliminating the communication between bots and their botmasters effectively means turning off the botnet. This led to the development of the decentralised configuration discussed next.

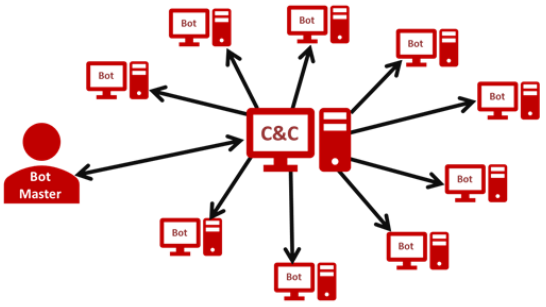


FIGURE 2.2: Centralised Topology

Decentralised Topology (Peer2Peer/Random/No C&C)

The Peer-to-Peer topology, or P2P for short, is another botnet configuration that exists (see Figure 2.3). In this architecture, as its name suggests, bots can have a control role in addition to their usual role. In more detail, the bots can communicate directly with each other so that if the botmaster is removed (or some of the bots are removed), the botnet continues to function. This means that it is no longer necessary to communicate with the botmaster which gives the advantage of being resilient to failure. Therefore, identifying bots does not necessarily mean turning off the entire botnet as is the case with the centralised topology.

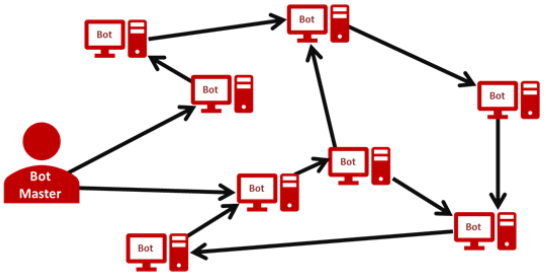


FIGURE 2.3: Decentralised Topology (P2P)

In addition, the P2P architecture offers more flexibility and robustness especially when the number of bots is large. The flexibility stems from their ability to exist on machines and communicate with other bots directly without the need to communicate with a

central point. There are a few techniques that are used to construct a P2P botnet. The process usually has two steps. In the first step, peer candidates need to be selected. And in the second, actions need to be implemented so the selected candidates become members of the botnet. Examples of bots that adopt the P2P configuration for communication are *Nugache* (Stover et al., 2007) and *Sinit* (Wang et al., 2007).

The Hybrid C&C model

In this architecture, functionalities from both centralised and decentralised botnets are used. In general, bots that are part of a hybrid P2P botnet can be either server bots or client bots. The server bots exhibit the behaviour of both clients and servers whereas the client bots are configured to act as clients only.

Random C&C model

In this architecture, the bot does not continuously communicate with the botmaster or other bots, rather it actually waits for the botmaster to make connection attempts. For an attack to be performed, the botmaster tries to connect to idle bots, if it finds any, it sends them commands to perform attacks. This model is not too difficult to implement and it is not easy to identify and interrupt because the communication between the bot and botmaster is not initiated by the bot. On the other hand, coordination issues can arise if the number of bots is large as the botmaster has to go through a large list of bots. There are no real botnets that use this model, it is only theoretical as it was suggested by (Cooke et al., 2005).

2.1.2 Botnet Examples

Many different botnets have appeared over the years. They differ in several aspects such as the topology, the main task to perform, the botnet size and so on. Some of these botnets are evolved and changed over time to become harder to detect and interrupt. It is important to point out that malicious internet applications are sometimes categorised according to their function. In other words, many malicious applications such as ransomware have a botnet architecture (or an architecture similar to that of botnets). They have client tools that reside on victim hosts and respond to commands coming from a central point. Therefore, the work in this thesis can be applied to detect any of these malicious applications with botnet architecture via network traffic analysis. The following subsections provide a high-level overview of some botnets and malware (i.e. malicious software with botnet architecture) examples:

2.1.2.1 SDBot

This family of bots is common, as its original developer has made it publicly available (i.e. open-source). This has led to it being developed into several flavours and variants. This botnet spreads itself through network shares that use empty or easy to guess passwords (Kharouni, 2009). It is noteworthy that this botnet, or one of its variants, can appear under several names such as backdoor.*Sdbot, Troj/Sdbot, BKDR_SDBOT or Backdoor.IRC.Sdbot*. When infecting devices, this botnet connects to a vulnerable device and executes a script to download itself into that device to infect its underlying system. After that, it opens a backdoor to enable the attacker to take control of that system.

2.1.2.2 RBot

This family of bots is known to be complex and hard to interrupt (Dietrich et al., 2011). It has hundreds of variants and these variants can have different names and techniques. This family of bots had a significant impact on how botnets avoid identification as it was the first family to use encryption or compression algorithms in its communication. Systems infected by RBot can be controlled and use to participate in DDoS attacks, key logging, spamming and so on. Like SDBot, RBot can have several names such as *W32.Spybot.worm, Worm_RBot or Backdoor.RBot.gen.* RBot easily infects systems with blank or weak passwords (much like SDBot). In addition, it targets some well-known flaws in the Windows Operating System. It is also interesting that some variants of this botnet can exploit backdoors or open ports created by other botnets.

2.1.2.3 Zeus

Zeus (or Zbot) is one of the peer-to-peer botnets that first emerged in 2007 (Binsalleeh et al., 2010). This family of botnets is mainly used for stealing money (or cyber fraud in general), phishing, banking information stealth and other attacks. The Zeus botnet was designed to steal information through man-in-the-browser attack via key-logging techniques and forms grabbing. Its method for spreading is through drive-by-download (e.g.**email, network, websites, torrent . . . etc.**) and phishing (e.g.*emails, webchat*). Usually, Zeus's malicious code is hosted on a site and when a user visits that site, their device is infected. The same can happen when the site displays an advertisement instead of actually hosting the code. When a device is infected, it joins the Zeus botnet and it can be under the attacker's control.

2.1.2.4 WannaCry

WannaCry is one of the most recent botnets at the time of writing this thesis. It is reported to have affected organisations in 150 countries (Kalita, 2017). Although it is usually classified as ransomware, because it demands payment after launching the cyber attack, it is effectively a botnet. According to many computer security experts, WannyCry uses a flaw in software that was developed by Microsoft. When a device is infected, WannaCry locks, or encrypts, the files and demands a quick payment (which increases with time). In more detail, it makes all the data on the infected computer system inaccessible (by locking it) and only allows the user to access two files; one of which contains instructions on what to do next and the other is the WannaCry tool itself. When the tool is launched it informs the computer users of the encryption of their files, and tells them that they only have a few days to make a payment. It also warns users that they will lose their files if they fail to make the payment. One interesting aspect of WannaCry is that it demands payment in bitcoin. It provides info on how to buy bitcoins and where to send them to.

2.2 What is Reverse Engineering?

Before delving into the details of the work carried out as part of this thesis (this is going to be explained in Chapter 3), it is better to define what is meant by reverse engineering. Reverse engineering can be defined as the practice of dismantling an object to examine, analyse or investigate its internal structure in order to improve it (Müller et al., 2000). This is not to be confused with re-engineering (Koschke, 2005) which is concerned with redesigning an object so that it becomes better in one or more aspects (i.e. to overcome the object's weaknesses or faults).

2.3 An Overview of Machine Learning and Transfer Learning

One of the main contributions of this thesis is *automatic botnet detection via transfer learning* (which is a machine learning technique). The data used to train machine learning models for machine is usually processed into a matrix structure with features (sometimes called attributes, descriptors or variables) as columns and instances (sometimes called examples or data points) as rows. Normally the class feature (sometimes called the target feature) is the last feature (i.e. the last column in the matrix). This Section provides a brief overview of *what machine learning is, how it works and why it is useful.* It also explains *what transfer learning is, how it works and when transfer learning can be more useful than traditional machine learning techniques.*

2.3.1 What is Machine Learning

Machine learning is mainly about developing and applying algorithms that can learn from data (Bishop, 2006). Primarily, the objective is to automatically explain the past and predict the future through data analysis. This field combines several other fields that include statistics, data science, artificial intelligence and database technologies. Hence, it is a multi-disciplinary field. It is worth mentioning here that data is a key component. Without data nothing practical can be done and only theoretical concepts and ideas can be developed at most (see Chapter 5 for more details).

Explaining the past is done via data analysis and exploration. Here statistical and visualisation techniques are usually used to describe the data. This is performed to highlight important relationships, patterns, trends or aspects that exist in the data so

further analysis can be performed. Furthermore, predicting the future is performed via modelling. In predictive modelling (Kuhn and Johnson, 2013), a model is created from the data in order to make a prediction. The prediction can be made for one or more outcomes. Here it will be simple and work will be done with a single outcome. If such an outcome is categorical, then the process is called *classification*. If the outcome is numerical, then the process is called *regression*. If the predictive process is about grouping similar instances of the data into groups of similar instances then it is called *clustering*. There are more processes in machine learning and data mining but this brief overview should be sufficient as an introduction. *Note that the work presented in this thesis is focused on classification.* Making automatic predictions is highly regarded nowadays. The historical data that is already stored, and the large amounts of data that is generated everyday, can help businesses (via machine learning and data mining) derive valuable insights and knowledge (Wu et al., 2014). This extracted knowledge can help in decision making and future planning to improve efficiency and maximize gains and profits.

2.3.2 What is Transfer Learning

Human beings in general have the ability to utilise past learning experiences when they are faced with new tasks. The level of mastering the new task depends on how much it is related to the past task. In machine learning, the sub-field that attempts to apply this experience, or knowledge transfer is known as *Transfer Learning*.

As explained in (Torrey and Shavlik, 2009), in traditional machine learning algorithms one deals with tasks individually, meaning if one has several tasks he/she learns each one separately. By contrast, transfer learning attempts to learn one or more tasks (known as

source tasks) and use the knowledge learned to enhance learning in another task (known as the target task). The target and source tasks must be related in one way or another.

Transfer learning is typically employed when there is little, or limited, amounts of labelled data in one task (known as the target task), and plenty of data in another related task (known as the source task). The assumption here is that using only the target data will result in less accurate models since there is insufficient data. Whereas, by transferring knowledge from the source task to the target task the model quality can be improved.

There are three key research issues in transfer learning (Pan and Yang, 2010). The first issue is *what to transfer*, which is concerned with the parts of knowledge that can be transferred between tasks because not all knowledge is common between different tasks (i.e. some knowledge can be specific for specific tasks). It is worth mentioning here that the work in this thesis is focused on this issue. The second issue is *how to transfer*. This is related to whether to transfer the knowledge as is or to apply some form of modification such as using weights. The third issue is *when to transfer*. This is an important aspect of transfer learning as in may cases the knowledge transfer ends in a negative transfer. That is, instead of improving learning in the target task, the knowledge transferred deteriorates the process.

Transfer learning techniques can be categorised into three sub-settings. *Inductive* transfer learning, *transductive* transfer learning and *unsupervised* transfer learning. The focus here will be on Inductive transfer learning, the reader is referred to the survey in (Pan and Yang, 2010) for more details on the other two categories. The work in this thesis is focused on inductive transfer learning because the data comes from a similar domain (network traffic) and all the datasets used have the same feature space (i.e. they

have the same feature sets). This similarity in domain and features (notice features and not necessarily feature values) makes the research problem automatically falls under the umbrella of induction because one needs to find where the differences are and how to exploit them.

2.3.2.1 Formal Definition of Transfer Learning

After providing a textual definition of transfer learning, this section formalises the problem and presents it in an intuitive way. This formalisation is based on the survey in (Pan and Yang, 2010). The definition breaks down the problem into its basic components such as Domain, Task and so on.

Domain and Task:

Let us assume that there exists a feature space $\mathcal{X}$ and a marginal probability distribution (for data in that space) $P(\mathbf{X})$, where $\mathbf{X} = \{\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots \mathbf{x}^{(n)}\} \in \mathcal{X}$.

A Domain $\mathcal{D}$ can be defined as the combination of the feature space $\mathcal{X}$ and marginal probability distribution $P(\mathbf{X})$ as follows:

$$\mathcal{D} = \{\mathcal{X}, P(\mathbf{X})\}$$

In addition to that, let us assume that there is a label space $\mathcal{Y}$ (that contains all possible labels for all instances in data $\mathbf{X}$) and a predictive function $f(\bullet)$ which is unknown. The purpose of this function is to predict a label that is in $\mathcal{Y}$ given input data.

A Task $\mathcal{T}$ can be defined as the combination of the label space $\mathcal{Y}$ and the predictive function $f(\bullet)$ as follows:

$$\mathcal{T} = \{\mathcal{Y}, f(\bullet)\}$$

The predictive function $f(\bullet)$ can be learned from the training data of the form $\{\mathbf{x}^{(i)}, y^{(i)}\}$, where $\mathbf{x}^{(i)} \in \mathbf{X}$ and $y^{(i)} \in \mathcal{Y}$, and used to predict label $y^{(i)}$ for data point $\mathbf{x}^{(i)}$ ($f(\mathbf{x}^{(i)}) = y^{(i)}$).

Source and Target Domain and Task:

It was mentioned previously that transfer learning is focused on learning from one or more source tasks to augment learning in a target task. To represent this using the above notation, the following two domains can be defined:

1. **Source domain:** $\mathcal{D}_S = \{\mathcal{X}_S, P_S(\mathbf{X})\}$ where $\mathbf{X} = \{\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots \mathbf{x}^{(n)}\} \in \mathcal{X}_S$
2. **Target domain:** $\mathcal{D}_T = \{\mathcal{X}_T, P_T(\mathbf{X})\}$ where $\mathbf{X} = \{\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots \mathbf{x}^{(n)}\} \in \mathcal{X}_T$

And Similarly, two corresponding tasks:

1. **Source task:** $\mathcal{T}_S = \{\mathcal{Y}_S, f_S(\bullet)\}$ where $f_S(\bullet) \rightarrow y^{(i)} \in \mathcal{Y}_S$
2. **Target task:** $\mathcal{T}_T = \{\mathcal{Y}_T, f_T(\bullet)\}$ where $f_T(\bullet) \rightarrow y^{(i)} \in \mathcal{Y}_T$

Observe that the predictive functions connect the source and target domains to the source and target tasks respectively as follows:

$f_S(\mathbf{x}^{(i)}) = y^{(i)}$ where $\mathbf{x}^{(i)} \in \mathcal{X}_S$ and $y^{(i)} \in \mathcal{Y}_S$

$f_T(\mathbf{x}^{(i)}) = y^{(i)}$ where $\mathbf{x}^{(i)} \in \mathcal{X}_T$ and $y^{(i)} \in \mathcal{Y}_T$

Transfer Learning:

Given a source domain $\mathcal{D}_S$ and learning task $\mathcal{T}_S$, a target domain $\mathcal{D}_T$ and learning task $\mathcal{T}_T$, the purpose of transfer learning is to enhance the learning of the target predictive function $f_T(\bullet)$ in $\mathcal{T}_T$ using the knowledge $\mathcal{D}_S$ and $\mathcal{T}_S$, where the source and target

domains are different (i.e. $\mathcal{D}_S \neq \mathcal{D}_T$), or the source and target tasks are different (i.e. $\mathcal{T}_S \neq \mathcal{T}_T$). Given this definition, there are multiple scenarios and cases:

- **Scenarios 1:** When the source and target domains are different (i.e. when $\mathcal{D}_S \neq \mathcal{D}_T$) at least one of the following conditions is satisfied:
  1. $\mathcal{X}_S \neq \mathcal{X}_T$ (feature spaces are different)
  2. $P_S(\mathbf{X}) \neq P_T(\mathbf{X})$ (probability distributions are different)
- **Scenarios 2:** On the other hand, when the source and target tasks are different (i.e. $\mathcal{T}_S \neq \mathcal{T}_T$) at least one of the following conditions is satisfied:
  1. $\mathcal{Y}_S \neq \mathcal{Y}_T$ (label spaces are different)
  2. $f_S(\bullet) \neq f_T(\bullet) \Leftrightarrow P_S(\mathbf{y}_S|\mathbf{X}_S) \neq P_T(\mathbf{y}_T|\mathbf{X}_T)$ (predictive functions are different)

It is important to notice that work in this thesis falls under the second case in the first scenario.

2.3.2.2 Inductive transfer learning

In transfer learning in general, there are two different tasks (i.e. the source and target tasks are not the same) coming either from the same domain or from two different domains. In inductive transfer learning, the most important thing is to have different source and target tasks, it does not matter if the source and target domains are different or the same. For example, under the domain of textual article classification, one task can be identifying sports articles and another task can be identifying politics articles. Here, although the domain is the same and the tasks are different, transfer learning between

them is still possible. Inductive transfer learning can be performed in more than one way:

- **The instance-transfer approach:** This approach says, let us not borrow all data from the source task directly, but rather, let us try to use relevant instances from the source task, along with data in the target task, in building a model for the target task. An example approach is TrAdaBoost that can be found in (Dai et al., 2007). TrAdaBoost is based on the classical AdaBoost algorithm. It works when the source and target tasks have the same set of features, but different data distributions. In addition, TrAdaBoost assumes that some of the data in the source task can be useful (i.e. leads to positive transfer) and some can be harmful (i.e. leads to negative transfer). The idea is to assign weights to data from the source task in such a way that useful data can have more effect than harmful data. The author has made the java implementation of this approach publicly available.

- **Feature representation transfer:** In this approach, attempts are made to find feature representations that reduce classification error. This task is also known as common feature learning (Argyriou et al., 2008). Methods for feature representation transfer can be supervised (Argyriou et al., 2007, Lee et al., 2007) or unsupervised (Raina et al., 2007).

- **Parameter-transfer:** In this approach it is assumed that models created for individual related tasks should have some parameters in common. Some of the existing approaches transfer parameters of Support Vector Machines (Evgeniou and Pontil, 2004) and priors of Gaussian Processes (Lawrence and Platt, 2004).

- **Relational-knowledge transfer:** Methods falling under this approach focus on transfer learning in relational domains. Approaches employ techniques from

statistical relational learning to transfer relationships from the source to target domains. An example method is reported in (Mihalkova et al., 2007) where relational knowledge is transferred across relational domains by using Markov Logic Networks (Richardson and Domingos, 2006).

2.4 Existing Work on Android Botnet Detection

Many approaches for botnet detection have been reported in the literature. The work in (Feng et al., 2014) introduced an approach that is based on the analysis of network traffic. They extract a set of features (i.e. attributes) from traffic chunks and then use machine learning algorithms to identify whether the traffic is malicious or not. Other works include BotMiner (Gu et al., 2008), BotHunter (Gu et al., 2007) and more recently BotDet (Ghafir et al., 2018). As the work presented in this chapter is focused on botnet detection on the Android operating system, the key existing approaches will be summarised in the remainder of this section. An interesting approach is Dendroid (Suarez-Tangil et al., 2014) where malware Android apps were grouped into families by analysing their source code. A similarity measure was used to taxonomise apps and create a phylogenetic-tree like structure. One approach to detect malicious Android apps was the work in (Sheen et al., 2015). In this work, features such as the API calls and permission requests that an APK file makes are used in separate datasets and an ensemble of classifiers (collaborative decision fusion) was used to perform predictions. Another approach that is related to permissions can be found in (Wei et al., 2015). In their work, they use techniques from the text mining domain to analyse the relationship between permission requests that an Android app makes and its textual description.

Although it focuses on malware detection, the approach in (Yerima et al., 2013) reverse engineers Android apps and analyses their source code. It focuses on detecting API calls made by the apps as well as the permissions that apps require and the commands that apps execute. The idea here is that by examining whether known APIs are being called from within an app's source code, it is possible to determine the app's intended behaviour during runtime. This is the same with permissions requested by apps and commands that apps attempt to run. In general, the approach builds a profile for each app by extracting these features and then uses a Bayesian classifier (e.g. NaiveBayes) to predict the probability of whether a new unseen Android app is suspicious or benign.

Another malware detection and classification technique is the method in (Kang et al., 2016) which disassembles malware Android apps to obtain and use their opcodes for malware detection. It uses n-opcode information to generate two different types of representation. One representation was the binary n-opcodes which describes the n-opcodes that have been used in an application, whereas the other representation was frequency n-opcodes which contains the counts of n-opcode in an application. This approach evaluates the performance of several classical machine learning classifiers such as Support Vector Machine (SVM) and RandomForest (RF). An interesting finding of this approach was that a high accuracy was obtained when using a small value for n in frequency n-opcodes.

In addition to the previous work, another method that disassembles malware Android apps to obtain and use their opcodes for malware detection is reported in (McLaughlin et al., 2017). One hot encoding is then used to create feature vectors based on the obtained opcodes. These vectors are then fed into a convolutional neural network (CNN) which learns the intrinsic characteristics of the data and yields a high classification accuracy.

Several approaches use static or dynamic analysis techniques. In static analysis, attempts are made to detect malicious activities without the need to execute the Android apps. The main idea is to model how the Android apps work by constructing and analysing some graphical models. An example of this type is the recent work in (Junaid et al., 2016) where an approach is presented to detect malicious behaviour in Android apps using models of their life cycles. Reverse engineering was used in this approach to construct a life cycle model for each Android app. After that, possible event sequences are derived from these models and used in attack detection. They developed a system called Dexteroid to identify SMS (Short Message Service) when they are sent to costly numbers as well as whether sensitive data is being leaked. Other examples include the work carried out in (Gordon et al., 2015) who built a tool called DroidSafe, the work in (Arzt et al., 2014) who built a tool called FlowDroid and the work in (Yang and Yang, 2012) who developed a tool called LeakMiner. On the other hand, approaches that employ dynamic analysis try to execute the Android apps to perform specific tasks and use the resulting data to detect malicious attacks. Some recent examples are the works in (Bai et al., 2016, Yan and Yin, 2012, Yang et al., 2013). Additionally, the authors of (Yang et al., 2015) proposed an approach to build data flow models from the reverse engineered source code of Android apps. Their method tries to build data flow models by detecting where data enters an application and how this data moves through it. In other words, they build trees of classes, methods and variables and use these trees to identify malicious code.

An approach, denoted DynaLog, is presented in (Alzaylaee et al., 2016, 2017) to generate dynamic features for malware detection. This approach attempts to overcome obfuscation techniques (used in malware families to avoid detection) by extracting features that describe the behavior of malware rather than its source code or contents.

DynaLog exploits open source tools to log and derive several low level events which gives it the advantage of being able to inspect the behaviour of Android apps at a deeper level. Once these logs are generated, several informative features are extracted and used in the identification process.

A recent approach that performs reverse engineering to obtain the source code of Android apps and analyse it is reported in (Kabakus and Dogru, 2018). In this work, the obtained source code was then analysed for API calls which was used as an indicator of the maliciousness of an app.

In general, source code analysis has been used before for the purpose of malicious code detection. For example, the work in (Benjamin and Chen, 2013) uses genetic algorithms for feature selection in an attempt to analyse the behaviour of malicious apps. The developers of this technique collected a total of 770 malicious applications written in several programming languages and used text mining techniques to transform the source code into data suitable for data mining techniques. The extracted features were used as input to the genetic algorithm. The effectiveness of selected feature subsets was later assessed.

Another recent approach that is has a degree of similarity to the approach discussed in this chapter can be found in (Nikola et al., 2017). The authors report two techniques: an app permission analysis technique and a source code analysis technique. The idea behind the second technique was to reverse engineer Android apps and automatically inspect their source code using text mining and machine learning methods and algorithms. The main focus of the work in (Nikola et al., 2017) was to detect malicious parts of the source code. In other words, to analyse the entire source code of an app and attempt to spot the code sections where malicious activities are carried out. Although this is indeed an interesting idea, the approach presented in this chapter is more focused on

making a precise prediction on the entire code of an app as a whole. Another problem regarding the approach in (Nikola et al., 2017) is that the data that was used seems to be no longer available. Therefore, it is not clear whether they have used malware from different families or several variations of the same malware. In addition, another point is that it is not clear whether the normal apps used are network apps. It is worth highlighting that one of the noticeable aspects of the work presented in this chapter is that it uses botnets from different families and not variations of the same botnet. Furthermore, the normal apps employed were network apps. More information on this is provided in Section 3.2.

2.5 Recent Work on Network Traffic based Botnet Detection

There is a large number of existing approaches that attempt to detect and identify botnets using network traffic. A good survey reporting many of these attempts can be found in (Silva et al., 2013).

The approach in (Garcia and Pechoucek, 2016) analyses network traffic by using a graph representation to represent connections made using botnets. It creates a graph for each Source IP where nodes contain tuples representing destination IP, destination port and the protocol. The edges of such graphs represent flows between nodes. Then the nodes and edges of the graph are updated in such a way to reflect how many times each a node and a graph is repeated as well as when a node makes a self loop.

In (Stiborek et al., 2018) a botnet detection approach that uses the interactions of malicious traffic with system resources as a data representation was proposed. The idea

is based on defining a similarity measure to reflect the properties of various resource types. It uses the same vocabulary concept that is commonly used in multiple instance learning (MIL). After that a clustering algorithm is used to group the data in separate groups.

Analysis of traffic flow characteristics was performed in (Kirubavathi and Anitha, 2016) to detect malicious traffic. The method attempts to extract multiple high impact features from network flows and then employs classical learning algorithms to classify data. The authors report that the features extracted are irrelevant of the packet contents which makes the method suitable for analysing encrypted traffic. Another technique that focuses on extracting significant features from network traffic is reported in (Bartos et al., 2016).

Combining multiple classifiers (i.e. ensemble) was used in (Bijalwan et al., 2016) to analyse network traffic data and attempt to detect botnet traffic. The authors only mention that they used an a freely available PCAP dataset, extracted features and evaluated an ensemble of classifiers. It is not clear what they used to extract features or what these features are.

Artificial Neural Networks (ANNs) were used for network traffic analysis as discussed in (Beghdad, 2008). They provide a technique to model user behaviour and therefore they can be seen as a suitable technique. The technique used in (Abuadlla et al., 2014) employs ANNs for network data analysis where a two-stage method is proposed. The technique uses a neural network of type multi-layer feed-forward (MLFF) where possible attacks are detected in the first stage (i.e. to identify whether the traffic is Normal or Malicious) and, after that, an attempt is made to identify the attack type in the second stage.

Another ANNs method was proposed in (Jadidi et al., 2013) where a multi-layer perceptron (MLP) was used in an attempt to spot suspicious network traffic. The MLP used in this technique used two different optimisation techniques to optimise weights between neurons. These techniques are the Cuckoo (Rajabioun, 2011) and Particle Swan Optimization with Gravitational Search Algorithm (PSOGSA) (Mirjalili and Hashim, 2010). According to the experiments conducted in this work, the PSOGSA yields better results than Cuckoo. ANNs have also been used in hardware-based detection systems. an example is the method reported in (Tran et al., 2012) where a block-based neural network (BBNN) was built using a field-programmable gate array (FPGA). A genetic algorithm was used to optimise the interconnection weights with the purpose of obtaining the best possible detection and false alarm rates.

Another machine learning technique that was used for network traffic analysis and malicious traffic detection is support vector machine (SVM). According the review in (Liao et al., 2013) it can be successful in many cases especially when analysing numeric data in binary classification (in other words detecting whether traffic belongs to one of two categories). One of the existing methods is the SVM-based approach reported in (Yuan et al., 2010). This technique applies feature selection using a discriminator selection algorithm to have the optimal feature subset. After that the data is used train an SVM classifier which is later employed to predict the category of unseen traffic. One-class SVM (OC-SVM), which is a technique used for anomaly and outlier detection, was used in (Winter et al., 2011) to detect malicious network traffic. Only malicious data was used to train the OC-SVM learner and then used to isolate similar patterns in future data. The authors claim that this approach is *inductive* because it can recognise, not only the patterns it has seen before, but also their variations.

Another popular machine learning technique that was used in analysis of network traffic

is the K-Nearest Neighbors (KNN) classifier. This algorithm is known to be simple to understand and easy to use because it classifies a new data point by a majority vote of its nearest training points (i.e. its neighbors). One of the existing techniques that is based on KNN is reported in (Costa et al., 2015). This technique builds a graph to perform Optimum-Path Forest Clustering (OPFC). The graph is based on KNN to assign weights to nodes because more than one optimisation technique was used to find the optimal value of k. As this can be considered a clustering technique, its results were compared to those of k-means and Self-Organising Maps (SOM) algorithms. The technique in (Abdulla et al., 2014) uses KNN with fuzzy logic to identify malicious traffic. Fuzzy logic was used to select labels for new instances whereas KNN was used to select the classes that are likely to match the real class value.

Decision trees (DTs) are known to be easy to interpret machine learning algorithms that have been successfully used in several areas. Hence, it comes as a no surprise that they have been utilised in network traffic analysis and recognition. One of the recent approaches can be found in (Rai et al., 2016). The developers of this approach propose a new method for selecting the value used in a node to make a split (i.e. a new branch of the DT is created based on the split value). According to the authors, the algorithms performs information gain based features selection and then selects a split value that ensures that the classifier is not biased towards prevalent values. The work reported in (Haddadi et al., 2014) uses DTs and genetic programming for network traffic analysis and classification. The authors indicate that their method can be used even when packet payload data is encrypted. The features used in this work were extracted from only the packet header information (which means the technique should work regardless of the payload being encrypted or not). The developers used an open source tool called *Softflowd* to extract two sets of features which were later analysed in an attempt to

develop an understanding of differences between botnets and their behaviour. It is noteworthy that Random Forest has also been used in network traffic analysis. Random Forest is based on the idea of sampling the input data several times, generating various DTs using the sampled datasets and then combining the predictions of these DTs. One of the existing techniques can be found in (Zhang et al., 2008).

Another network traffic analysis technique that uses DTs is reported in (Zhao et al., 2013). The authors mentioned that they wanted to select a classifier that adapts to real time data changes and, after investigating various machine learning algorithms, they chose to work with DTs. After some analysis, the authors report that they worked with the Reduced Error Pruning algorithm (REPTree) because it enhances the detection accuracy when data is noisy and the resulting model is usually small which reduces complexity.

An existing approach that evaluates the performance of several classical classifiers can be found in (Stevanovic and Pedersen, 2014). The authors report that they experimented with eight commonly used classifiers which included ANNs, SVM and Random Forest. The results reported in this work show that the performance of different classifiers can vary as only two of the used eight exhibited promising results. In fact, the conclusion was that tree based learners (i.e. Random Forest and Random Tree) performed better than others classifier families.

A recent system that focuses on identifying command and control (C&C) traffic is BotDet which is discussed in (Ghafir et al., 2018). BotDet contains as many as four techniques to detect C&C traffic. These techniques includes an implementation of a module that uses a predefined list of known malicious IPs of C&C servers to identify connections to those servers. Another module is based on a black list of secure socket

layer (SSL) certificates which are known to be malicious as opposed to IPs. SSL certificates are used by malicious applications to encrypt communication and make such applications harder to detect. BotDet also includes a module that attempts to detect any connections to a Tor network. In addition, it contains a module that prevents hosts from using the domain flux technique which enables infected hosts to connect to undesired domain name servers. In general, BotDet is an implementation and realisation of already existing approaches.

The first attempt to use transfer learning in network traffic classification was introduced in (Zhao et al., 2017) where feature transfer learning was used, as opposed to the method proposed in this thesis which is instance transfer. The technique is based on projecting the source and target data into a common latent shared feature space and then using this new feature space for making predictions. The technique works in such a way that it attempts to preserve the distribution of the data. Although this the results reported by the author seem to be reasonable, there is no freely available tool or code to use for comparison. As this technique is iterative, it is computationally heavy. The approach proposed in this work is different as it performs instance transfer by performing only one pass over the target data.

A recent work that applies transfer learning for classification of network traffic can be found in (Sun et al., 2018). This work does not propose a new transfer learning method, rather, it only evaluates the performance of an existing open source transfer learning algorithm called TrAdaBoost (Dai et al., 2007). Although the results show performance improvement when compared against the base classifier without transfer (referred to as NoTL in the publication), it is noteworthy to mention that TrAdaBoost was extended and enhanced by the introduction of TransferBoost (Eaton and desJardins, 2011) (which

is the algorithm that is used for evaluation and comparison of results as will be explained in more detail in Sections 4.2 and 5.4).

Instance transfer learning has been applied in multiple areas. For example, the recent work in (Liu et al., 2018) reports an attempt that employs Multiple Instance Learning (MIL) in text classification. This is a two stage method where, in the first stage, the algorithms decides whether the source and target tasks are similar enough to perform transfer which leads to the second stage where transfer is performed.

Note that the methods presented in this thesis differ from existing technique. For example, the Android botnet detection is a proactive approach that attempts to detect botnet apps before they are executed. In other words, it tries to detect danger before it occurs. This is performed by reverse engineering and analysing the source code of an Android app. More details about this technique are presented in Chapter 3. In addition, this thesis presents a novel transfer learning approach for the automatic detection of botnet families by means of network traffic analysis. This transfer learning approach is based on measuring the similarity of instances in source and target datasets and transferring only instances that are deemed similar. This is based on the assumption that similar instances can have similar characteristics which means they have a potential to enhance models created using the target data after transfer. This thesis does not only present a new approach, but it also evaluates and compares its performance against other existing commonly used approaches. In addition, this thesis discusses a limitation of this approach and provides an extension that overcomes this limitation. This new transfer learning approach is discussed in detail in Chapters 4 and 5. In general, the contributions of this thesis are listed in Section 1.6.

2.6 Summary

This chapter was dedicated to background and literature review of related works in botnet detection and transfer learning. The first part of the chapter explained how botnets work, their architecture and the anatomy of their attacks. It also provided botnet examples. In order to grasp the work done in this thesis, it is important to have at least a high level overview of machine learning. The second section of this chapter represents an introduction to what machine learning and data mining is about. At the beginning, the section provided a brief definition of machine learning. After that, it described transfer learning, a machine learning sub-field, that is the focus of this thesis. The third part of this chapter summarised the most recent botnet detection approaches that use machine learning algorithms as their method of detecting malicious traffic. This part included a summary of Android botnet detection techniques.

Chapter 3

An Integrated Source Code Mining Approach for Android Botnet Detection

Android is one of the most popular smartphone operating systems. This makes it one of the default targets for malicious cyber-attacks. Android's Play Store is not very restrictive which makes installing malicious apps easy. As botnets are amongst the most dangerous cybercrime tools that are used nowadays on the internet, it is not surprising for botnet developers to target smartphone users and install their malicious tools on a large number of devices. This is often done to gain access to sensitive data such as credit card details, or to cause damage to individual host or organisation's resources by executing denial of service attacks. The main contribution of this chapter is that it proposes an approach to identify mobile (Android) botnet apps by means of source code mining. The source code is analysed via reverse engineering and data mining techniques for several examples of malicious and non-malicious apps. Two approaches are used

in this work to build datasets. In the first, text mining is performed on the source code and several datasets are constructed and, in the second, one dataset is built by extracting source code metrics using an open-source tool. After building the datasets, several classification algorithms are evaluated and their performance was assessed. It is worth mentioning here that several sections of this chapter were published by the author of this thesis in (Alothman and Rattadilok, 2017).

3.1 Overview

Android is a popular operating system that is now growing among smartphone users. This operating system is open source, customisable and user-friendly. Also, it is relatively easy to write Java applications that run smoothly on the OS. With the huge number of existing Java apps, it becomes difficult to know whether a new app is safe to install or not. To overcome this issue, among other challenges, it would be useful to automate the process of checking how safe a new app is (i.e. to use the smartphone itself to predict whether or not an app is safe).

In this work, an attempt is made to solve this problem by automatically reverse engineering Android apps, obtaining their Java source code and using the source code to make predictions. This is achieved by using data mining techniques to analyse the Java source code of these apps and try to predict whether a given app is bot or not. A summary of related work and existing approaches was given in the previous Chapter (Section 2.4).

The motivation and objectives of this work are to investigate using a *proactive* solution for Android botnet detection. The proposed solution tries to identify botnet apps before they are executed in order to eliminate harm before it occurs. This is achieved by

proposing, implementing and evaluating a source code mining method as will be explained in detail in subsequent sections of this chapter.

Observe that these apps are originally available as Android Application Package (APK) files and they are transformed into a format that can be automatically analysed. As part of this work, a collection of botnet and safe, or normal, apps was gathered (the botnet apps were obtained from the ISCX dataset (Abdul Kadir et al., 2015, Gonzalez et al., 2015) which is freely available on the Canadian Institute for Cybersecurity's website[1]).

The Dex2jar tool (Team, 2016) was employed to reverse engineer these Android apps and convert them into Java source code. In total, a collection of 21 apps was obtained (9 botnets and 12 safe) and these apps were used to create datasets suitable for data mining.

In these datasets, each app is an instance (i.e. example). Several attributes (i.e. features) were extracted for each app from its Java source code, and the class variable was either botnet (positive) or not (negative). After building these datasets, several classifiers were used and their performance was evaluated.

To the best of this thesis's author's knowledge, this is the first work to identify **Android botnet apps** by directly mining their source code. Therefore, the contributions of this work can be summarised as follows: This approach uses data mining techniques to analyse the Java source code in two ways. In the first method, the Java source code is treated as if it is normal text by using Natural Language Processing (NLP) methods (Weiss, Indurkhya and Zhang, 2004). And in the second approach, several statistical measures are extracted from the source code and used as attributes (i.e. features) in the dataset. This approach can be considered static as it does not require

[1]https://www.unb.ca/cic/datasets/android-botnet.html

the execution of the Android app itself. The idea is that as soon as an Android app is downloaded, it is reverse engineered and its Java source code is obtained and used to predict whether this app is safe or malicious. It is noteworthy that the advantage here is being *proactive*. In other words, it is an attempt to identify danger before it occurs.

Another point is that this work can be considered a behaviour-based approach as opposed to a signature-based approach. Using signature-based methods have the general disadvantage of relying on other people to report whether a certain app is malicious (signature-based methods work by comparing signatures, or hashes, of files or file contents on a system to a list of known malicious files).

In addition to the previous two points, another minor but key contribution of this work is that it makes sure the botnets used in experiments belong to different botnet families. This is highlighted because some approaches use variations of the same botnet to enrich data. For example, different versions of the same botnet app can be used as different examples (instances in the training data as will be shown later in this chapter). Another point this work ensures is that the normal apps used are network apps which makes the comparison and analysis more objective (i.e. it would be incompatible to compare botnets against local games or other apps that perform no network activities).

The remainder of this Chapter is organised as follows: Section 3.2 explains in detail how datasets used in this work were constructed. Section 3.3 has a short description of the algorithms which have been used in the proposed approach. Section 3.4 has the experimental results and discussions. The Chapter then ends with a summary.

3.2 Dataset Formation and Feature Extraction

As part of this work, reverse engineering was used to obtain the Java source code of the Android apps. As the APK files are compressed files, they were renamed to *.zip* and then unzipped which resulted in *.dex* files (Zhang et al., 2016). After that, the Dex2jar (Team, 2016) tool was used to convert the *dex* files into Java *jar* files. After tha Java jar files were obtained, the Java decompiler *JD-GUI* (Team, 2015) was used to regenerate the Java source code of the APK apps. The entire process is illustrated in Figure 3.1.

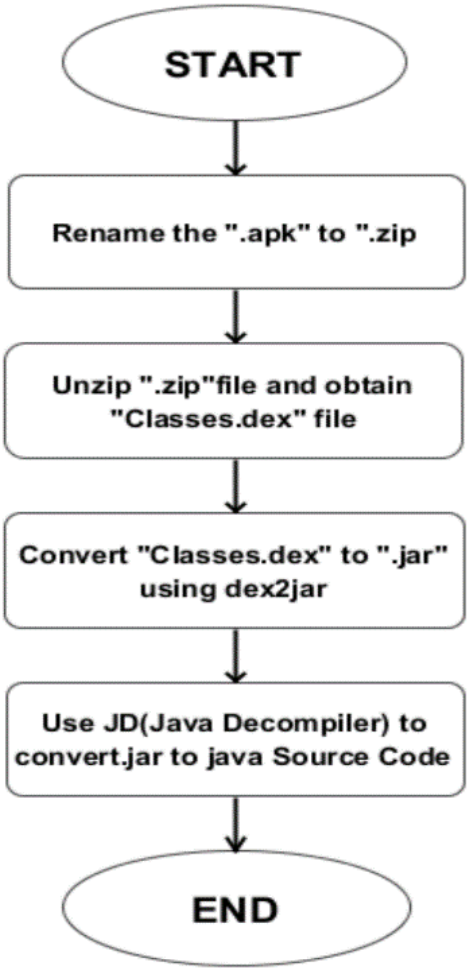


FIGURE 3.1: APK File Reverse Engineering

In order to make sure all apps analysed in this work are as similar as possible, only

network apps were used to represent *normal* apps (i.e. apps that are used to connect to and communicate with other users such as chatting and messaging apps). A list of the apps used in this work is provided in Table 3.1.

Botnet Apps	**Normal Apps**
Anserverbot	AndroIRC
Bmaster	SimpleIRC
DroidDream	Kik
Geinimi	LOVOO
Nickyspy	Line
PJapps	WhatsApp
Pletor	Hi5
Zitmo	SKOUT
Rootsmart	Viber
	Messenger
	WeChat
	SnapChat

TABLE 3.1: A List of the Apps used in the Experiments

To be able to predict whether a given app is botnet or not, predictive models are needed. For this purpose, the WEKA (Hall et al., 2009) open source machine learning platform was used (version 3.6.13). The methods used to create the datasets are explained in the following subsections.

3.2.1 Text-Mining Approach

Text mining (Witte et al., 2008) aims to process textual information, which is normally unstructured, and use the resulting structured data to build predictive models and to understand the original textual information better. The structured data is usually obtained by deriving numerical summaries about the documents based on the words they contain. To be able to use the Java source code (obtained after reverse engineering) to perform text mining, all the Java code of each app was concatenated into one file (which means there is now one large Java source code file per app) and then a dataset was created. This dataset initially has three columns: The app's name, the apps Java source code and the app's class (botnet or not).

After that, WEKA's TextToWordVector filter was applied with Term Frequency and Inverse Document Frequency (TF-IDF) on all the Java code. TF-IDF (Weiss, Indurkhya and Zhang, 2004) is a widely used transformation in NLP where terms (or words) in a document are given importance scores based on the frequency of their appearance across documents. This idea was used so that any information, such as Java class, method and variable names, or words used in comments, are assigned scores. A word is important and is assigned a high score if it appears multiple times in a document (i.e. Java source code of an app). However, it is assigned a low score (meaning it is less important) if it appears in several documents (Java code of several apps). WEKA's default parameters were used for this filter except for the *number of words to keep*. This parameter is 1000 by default, and it was changed to 3000 and 5000. This is because some of the apps had a large number of Java classes and lines. In more detail, the number of different words is expected to be high and therefore the value of this parameter was varied in an increasing order. An example dataset resulting after this process is shown in Table 3.2.

Observe that the features are Weights (W's) of words that result after applying the TF-IDF filter. And the dots ... mean so forth.

App Name	**W1**	**W2**	**...**	**Class (botnet or not)**
App 1	0.069	0.034	...	Yes
App 2	1.03	0.018	...	No
...	...	...	...	...
App n	0.009	0	...	No

TABLE 3.2: Dataset resulting after applying TextToWordVector and then TF-IDF filter

3.2.2 Source Code Metrics Approach

This method aims to use software metrics as characteristics (or features) of the Java source code obtained in Section 3.2. For this purpose, the tool CodeAnalyzer (CodeAnalyzer, 2017) was used and several quantitative measures were obtained. These include statistics such as the total number of files, the total number of code lines and the code to comment ratio. Table 3.3 provides an overview of these metrics.

Metric	**Definition**
Total Files (For multiple file metrics)	The total number of source code files in the project
Total Lines	The overall number of lines
Average Line Length	The average line length (sum of length of all lines divided by the overall number of lines)
Code Lines	The overall number of code lines
Comment Lines	The overall number of comment lines
Whitespace Lines	The overall number of empty lines
Code/(Comments + Whitespace) Ratio	The ratio of code lines to comment and empty lines
Code/Lines Ratio	The ratio of code lines to comment lines
Code/Comments Ratio	The ratio of code lines to comment lines
Code/Whitespace Ratio	The ratio of code lines to empty lines
Code Lines/File (For multiple file metrics)	The overall number of code lines divided by the total number of source code files in the project
Comment Lines/File (For multiple file metrics)	The overall number of comment lines divided by the total number of source code files in the project
Whitespace Lines/File	The overall number of whitespace lines divided by the total number of source code files in the project

TABLE 3.3: Source Code Metrics extracted by CodeAnalyzer

An example dataset resulting after this process is shown in Table 3.4. Observe that SCM stands for Source Code Metric.

App Name	**SCM1**	**SCM2**	**...**	**Class (botnet or not)**
App 1	33921	0.11	...	Yes
App 2	128998	0.21	...	No
...	...	...	...	...
App n	45635	0.33	...	No

TABLE 3.4: Dataset resulting after extracting Source Code Metrics

3.2.3 Feature Selection

WEKA's StringToWordVector with TF-IDF filter was applied with a various number of words to keep so that different sets of features (i.e. words) can be experimented with. This helps in obtaining more insight into the importance of different terms used in the source code, and at the same time helps in capturing the unimportant ones. In addition, using various numbers of words to keep helps in inspecting the effect of having sparse features because some terms might occur rarely and some others can be common. The resulting datasets had much more features than examples. For example, the number of features in dataset W3000 is 4332 (see Table 3.5) and the number of examples we have is 21. This means that, in this dataset, the number of features is more than 200 times the number of examples. Having a large number of features makes it practically impossible to interpret models and can cause overfitting (Liu and Motoda, 2007). Therefore, reducing the number of features can help avoid overfitting and build models which are easier to interpret and with better predictive performance. Having a smaller number of features can also reduce the computational time considerably (Liu and Motoda, 1998). As feature

selection tries to identify the most informative features and removes the uninformative, irrelevant, noisy or unreliable features, WEKA's SubSetEval feature selection algorithm was applied to each of these datasets. The selected features included words such as *lock*, *state* and *concurrent* and the removed features included words like *audio*, *recycle* and *widget*. Table 3.5 provides a description of all our datasets. Observe that these datasets are used in the experiments and evaluation performed in Section 3.4.

Dataset Name	**No of Features**	**Dataset Description**
Metrics	13	Resulted after extracting code metrics using CodeAnalyzer
W1000	1332	Resulted after applying WEKA's StringToWordVector with number of words to keep = 1000
W1000FS	24	Resulted after applying WEKA's SubSetEval feature selection algorithm to dataset W1000
W3000	4332	Resulted after applying WEKA's StringToWordVector with number of words to keep = 3000
W3000FS	85	Resulted after applying WEKA's SubSetEval feature selection algorithm to dataset W3000
W5000	7697	Resulted after applying WEKA's StringToWordVector with number of words to keep = 5000
W5000FS	21	Resulted after applying WEKA's SubSetEval feature selection algorithm to dataset W5000

Table 3.5: A Summary of the Created Datasets

3.3 Algorithms used in this Work

In this study, several machine learning algorithms in WEKA were used. They were selected because they are commonly used (Wu et al., 2007). The following Subsections provide a list of these algorithms and a brief introduction to each of them:

3.3.1 NaiveBayes

The Naive Bayes classifier (Rish, 2001) is based on Bayes theorem with independence assumptions between input variables (predictors). Suppose x was the input variables and c was the class, Bayes theorem introduces a method of calculating the posterior probability, $P(c|x)$, from $P(c)$, $P(x)$, and $P(x|c)$. These terms can be read as follows: $P(c|x)$ is the probability of the class c given the data x, $P(c)$ is the probability of the class c, $P(x)$ is the probability of the data (sometimes denoted the *evidence*) and finally $P(x|c)$ is the probability of the data x given the class c. The values of $P(c)$, $P(x)$ and $P(x|c)$ should be computable directly from training data. This classifier assumes that the effect of the value of an input variable (x) on a given class (c) is independent of the values of other input variables. This assumption is known as class conditional independence.

3.3.2 KNN

K-Nearest Neighbours (KNN) (Larose, 2004) is an algorithm that stores all available examples (i.e. instances) and classifies new examples based on a similarity measure (e.g. distance function). In more detail, the algorithm calculates the distance (or similarity) between an input example and the training examples and chooses the k examples that are closest (or more similar) to the input example. Then a majority vote of neighbours

is used to classify new examples. This is achieved by the choosing the most common class among the K-Nearest neighbours. In this study a value of five neighbours (K=5) was used. This is about a quarter of the total number of instances. Observe that an odd value, rather than even, was selected to avoid having ties (i.e. having a result where the number of neighbours that belong to one class is the same as the number of neighbours that belong to another class).

3.3.3 Decision Trees

Decision trees work is by building classification or regression models in the form of a tree structure (Rokach and Maimon, 2014). This is done by breaking down a dataset into smaller and smaller subsets while at the same time an associated decision tree is incrementally developed. The final result is a tree with decision nodes and leaf nodes. A decision node has two or more branches, and a Leaf node represents a classification or decision. WEKA provides more than one decision tree algorithms, in this work the J48 algorithms was used. J48 is a variation of the well-known C4.5 algorithm (Quinlan, 1993) which is decision tree. It was selected because it is a popular algorithm that is known to perform well in many areas and it ranked as the number one algorithm in the extensive experiments performed as part of a published evaluation (Wu et al., 2007).

3.3.4 RandomForest

This algorithm works by building many decision trees at training time and using them to vote for the class of a new example (Dua and Du, 2011). To construct each tree, the training data is obtained by randomly sampling both the examples and input variables (with replacement). In other words, because the training data is randomly samples in

terms of both examples and features, each decision tree in the RandomForest is trained with different data which gives different trees the ability to focus on different aspects of the data.

3.3.5 Sequential Minimal Optimization (SMO)

This is WEKA's implementation for the Sequential Minimal Optimization for Training Support Vector Machines (Platt, 1998). Support Vector Machines are a powerful technique that tries to find the plane (or hyperplane) which maximises the distance between points from different classes in vector space (Steinwart and Christmann, 2008). It mainly works in binary classification settings, but there are existing methods to make it work for multi classification and regression.

3.4 Experimental Results

The algorithms used in this study were introduced in Section 3.3. Each of these algorithms was run on each of the datasets described in Table 3.5. Five fold cross-validation was used to calculate a number of classification evaluation metrics (Santafe et al., 2015). Table 3.6 shows the average classification accuracy of each algorithm on each dataset. Accuracy (see Equation 3.1) can be defined as the percentage of predictions that a model gets right. Five-fold cross-validation was used because the dataset contains nine botnet apps and 12 normal apps. This will split the data into five equal parts making sure that each part contains botnet and normal apps (no app appears in more than one part). To evaluate an algorithm, it is trained on 4/5 of the data and an accuracy value is computed by testing the trained algorithm on the remaining 1/5. This is repeated five times, with a different testing part is selected

each time. Then the overall accuracy is calculated as the average of the five calculated accuracy values. It is worth mentioning here that these algorithms were ran with their default parameters unless mentioned otherwise.

Dataset	**NB**	**KNN(5)**	**J48**	**RF**	**SMO**
W1000	76.20%	81.00%	**85.70%**	**85.70%**	**85.70%**
W1000FS	85.70%	**95.20%**	85.70%	**95.20%**	**95.20%**
W3000	85.70%	**95.20%**	67.20%	90.50%	90.50%
W3000FS	90.50%	**95.20%**	85.70%	**95.20%**	**95.20%**
W5000	81.00%	**95.20%**	85.70%	90.50%	**95.20%**
W5000FS	85.70%	**100%**	95.20%	**100%**	**100%**
Metrics	**85.70%**	81.00%	67.20%	81.00%	81.00%

TABLE 3.6: A Summary of Performance Results (Average Accuracy of Classifiers on Various Datasets)

Some of the performance metric values for different algorithms in Table 3.6 and later tables in this chapter are the same. This is likely to be because of the small size of the dataset and number of folds selected for cross-validation. Different values can be obtained by using more data or a different number of folds).

$$Accuracy = \frac{n_{bot \to bot} + n_{nor \to nor}}{n_{bot \to bot} + n_{bot \to nor} + n_{nor \to bot} + n_{nor \to nor}} \times 100\% \tag{3.1}$$

According to Equation 3.1 accuracy in our experiments is calculated as the sum of the number of cases a model predicts correctly (i.e. the number of botnet instances which were correctly predicted as botnet added to the number of normal instances which were correctly predicted as normal) divided by the total number of cases (i.e. the number

of botnet instances which were correctly predicted as botnet added to the number of botnet instances which were incorrectly predicted as normal added to the number of normal instances which were correctly predicted as normal added to the number of normal instances which were incorrectly predicted as botnet). Observe that our class of interest is the *botnet* class and that these numbers are used to compute other metrics as follows:

The error rate, or ERR, is the fraction of predictions that a model gets wrong (Equation 3.2). The value of ERR is normally between zero and one (the closer to zero the better the classifier).

$$ERR = \frac{n_{bot \to nor} + n_{nor \to bot}}{n_{bot \to bot} + n_{bot \to nor} + n_{nor \to bot} + n_{nor \to nor}} \tag{3.2}$$

The False Positive Rate, or FPR, is the ratio between the number of botnet cases which were incorrectly classified as normal to the total number of botnet cases (Equation 3.3). The value of FPR is normally between zero and one (the closer to zero the better the classifier).

$$FPR = \frac{n_{bot \to nor}}{n_{bot \to nor} + n_{bot \to bot}} \tag{3.3}$$

The False Negative Rate, or FNR, is the ratio between the number of normal cases which were incorrectly classified as botnet to the total number of normal cases (Equation 3.4). The value of FNR is normally between zero and one (the closer to zero the better the classifier).

$$FNR = \frac{n_{nor \to bot}}{n_{nor \to bot} + n_{nor \to nor}} \tag{3.4}$$

The True Positive Rate, or TPR, is the ratio between the number of normal cases which were correctly classified as normal to the total number of normal cases (Equation 3.5). The value of TPR is normally between zero and one (the closer to one the better the classifier). TPR is also known as *Sensitivity*.

$$TPR = \frac{n_{nor \to nor}}{n_{nor \to bot} + n_{nor \to nor}} \tag{3.5}$$

The True Negative Rate, or TNR, is the ratio between the number of botnet cases which were correctly classified as botnet to the total number of botnet cases (Equation 3.6). The value of TNR is normally between zero and one (the closer to one the better the classifier). TNR is also known as *Specificity*.

$$TNR = \frac{n_{bot \to bot}}{n_{bot \to nor} + n_{bot \to bot}} \tag{3.6}$$

Another metric that we have calculated is the area under the curve, or AUC, which is often used in evaluating classifiers. Its main advantage is that, when several classifiers are used, it can be used to decide which of them is the best at predicting classes. The value of AUC is normally between zero and one (the closer to one the better the classifier). The reader is referred to (Bradley, 1997) for a comprehensive explanation of this metric.

The following tables show AUC, TPR, TNR, FPR, FNR and ERR for the five classifiers explained previously on all the datasets described in Table 3.5. The most interesting value of each metric is displayed in bold. Bear in mind that for some metrics we are looking for the minimum value and some others we are looking for the highest value.

Dataset	**AUC**	**TPR**	**TNR**	**FPR**	**FNR**	**ERR**
W1000	0.736	0.556	0.917	0.083	0.444	0.238
W1000FS	0.894	0.667	1.0	0.0	0.333	0.143
W3000	0.847	0.778	0.917	0.083	0.222	0.143
W3000FS	**0.903**	**0.889**	0.917	0.083	**0.111**	**0.095**
W5000	0.792	0.667	0.917	0.083	0.333	0.19
W5000FS	**0.903**	**0.889**	0.917	0.083	**0.111**	**0.095**
Metrics	0.843	0.778	0.917	0.083	0.222	0.143

TABLE 3.7: A Summary of NaiveBayes Results

Table 3.7 shows the results after using NaiveBayes. It can be seen that it equally scores the highest AUC and TPR on the two datasets W3000FS and W5000FS. On the other hand, it scores the highest TNR and lowest FPR on the W1000FS dataset. In addition, it equally scores the lowest FNR and ERR for on the two datasets W3000FS and W5000FS. It is interesting to see how NaiveBayes performs well on datasets generated using text mining techniques (especially after applying feature selection).

Dataset	**AUC**	**TPR**	**TNR**	**FPR**	**FNR**	**ERR**
W1000	0.958	0.667	0.917	0.083	0.333	0.19
W1000FS	0.931	0.889	1.0	0.0	0.111	0.048
W3000	0.935	1.0	0.917	0.083	0.0	0.048
W3000FS	1.0	0.889	1.0	0.0	0.111	0.048
W5000	0.949	1.0	0.917	0.083	0.0	0.048
W5000FS	**1.0**	**1.0**	**1.0**	**0.0**	**0.0**	**0.0**
Metrics	0.898	0.778	0.833	0.167	0.222	0.19

TABLE 3.8: A Summary of kNN Results

Table 3.8 shows the results after using the kNN classifier. It can be seen that it scores the best values for all metrics on the W5000FS dataset. This is probably because the increased number of features (i.e. number of words to keep) provides more information for the classifiers to distinguish between different classes. This is in addition to the fact that it performs equally well on some of other datasets.

Dataset	**AUC**	**TPR**	**TNR**	**FPR**	**FNR**	**ERR**
W1000	0.861	0.889	0.833	0.167	0.111	0.143
W1000FS	0.847	0.778	0.917	0.083	0.222	0.143
W3000	0.75	0.667	0.833	0.167	0.333	0.238
W3000FS	0.833	0.667	1.0	0.0	0.333	0.143
W5000	0.847	0.778	0.917	0.083	0.222	0.143
W5000FS	**0.944**	**0.889**	**1.0**	**0.0**	**0.111**	**0.048**
Metrics	0.741	0.556	0.917	0.083	0.444	0.238

TABLE 3.9: A Summary of J48 Results

Table 3.9 shows the results after using WEKA's J48 decision tree algorithm. It can be seen that it scores the best values for all metrics on the W5000FS dataset. This is in addition to the fact that it performs equally well on some of other datasets.

Dataset	**AUC**	**TPR**	**TNR**	**FPR**	**FNR**	**ERR**
W1000	0.977	0.778	0.917	0.083	0.222	0.143
W1000FS	0.986	0.889	1.0	0.0	0.111	0.048
W3000	0.981	0.778	1.0	0.0	0.222	0.095
W3000FS	0.995	0.889	1.0	0.0	0.111	0.048
W5000	0.972	0.889	0.917	0.083	0.111	0.095
W5000FS	**1.0**	**1.0**	**1.0**	**0.0**	**0.0**	**0.0**
Metrics	0.898	0.778	0.833	0.167	0.222	0.19

TABLE 3.10: A Summary of RandomForest Results

Table 3.10 shows the results after using WEKA's RandomForest algorithm. It can be seen that it scores the best values for all metrics on the W5000FS dataset.

Dataset	**AUC**	**TPR**	**TNR**	**FPR**	**FNR**	**ERR**
W1000	0.833	0.667	1.0	0.0	0.333	0.143
W1000FS	0.944	0.889	1.0	0.0	0.111	0.048
W3000	0.889	0.778	1.0	0.0	0.222	0.095
W3000FS	0.944	0.889	1.0	0.0	0.111	0.048
W5000	0.944	0.889	1.0	0.0	0.111	0.048
W5000FS	**1.0**	**1.0**	**1.0**	**0.0**	**0.0**	**0.0**
Metrics	0.819	0.889	0.75	0.25	0.111	0.19

TABLE 3.11: A Summary of SMO Results

Table 3.11 shows the results after using WEKA's SMO algorithm. The results show that SMO has the same TPR and FNR for all datasets.

For each dataset, the best performing algorithm was displayed in bold in Table 3.6 using the accuracy as a measure of performance. As it can be seen, some algorithms perform equally well for some datasets. The feature values are positive real numbers in all of these datasets. For the metrics dataset, the number of features is less than the number of examples. As for the other datasets, some of them had much more features than examples as can be seen in the No of Features column in Table 3.5.

The disadvantage of having much more features than examples was mentioned in Section 3.2.3 and the experimental results support what was discussed there. It is clear from the table that the performance improves significantly after applying feature selection. This case is true regardless of the number of features before applying feature selection (recall several values for the number of words to keep parameter were used when the TextToWord filter was applied). By analysing the experimental results further, an interesting observation can be made. That is, the performance of the Decision Tree (J48) algorithm is always the worst regardless of the dataset. Another observation is that the k-Nearest Neighbour algorithm (with five neighbours in these experiments) seems to be the best performing algorithm in general.

However, using other metrics for classification performance shows that results can vary. Looking at Tables 3.7, 3.8, 3.9, 3.10 and 3.11 reveals that in general the used classifiers perform well on dataset *W5000FS*. These classifiers showed high values for AUC, TPR and TNR, and low values for FPR, FNR and ERR. However, by observing the performance datasets generated using text mining methods, it can be noticed that performance improves as the number of features increases. In addition, it was interesting

to see that none of the classifiers performed best on the *Metrics* dataset and using feature selection of the datasets generating using text mining improves classifier performance.

Some might suggest that code obfuscation might be used to evade detection. It should be clarified that this is exactly what this proposed method is about. Even if a known botnet app uses such techniques, it just needs to be added to the training data and used in building predictive models. In fact, the code obfuscation techniques can provide additional features that can be added to the training datasets for use in building the predictive models. Another point that even if obfuscation is used to evade detection, the method proposed in this chapter can still be utilized after a de-obfuscation step has been applied. This means that appropriate de-obfuscation methods can be built into the preprocessing pipeline prior to the prediction stage. Lastly, if de-obfuscation fails, then the source code metrics approach can still be utilized since it is based on statistical methods that do not depend on the code syntax.

3.5 Summary

This chapter provided a complete overview of a new method that was developed for the detection of botnet Android apps. The method is based on reverse engineering Android apps and mining their source code via Natural Language Processing and statistical techniques. The chapter began by providing an overview of the problem and summarising major existing techniques. After that a detailed explanation of how the technique works, how the datasets were created and how the experiments were run was given. The chapter ended with analysing and discussing the results which showed that the proposed method can detect botnets with higher accuracy especially when feature selection is applied.

Chapter 4

A Novel Similarity-Based Instance Transfer Learning Approach for Botnet Family Classification

The previous chapters introduced several topics that are essential to understand how botnets work. Not only this, but they also contained an introduction to machine learning and transfer learning which are at the core of this thesis. In addition, a review of existing approaches was also conducted. This chapter explains in detail a novel transfer learning method for botnet detection via network traffic analysis. The novel algorithm is called Similarity-Based Instance Transfer Learning, or SBIT for short. The chapter explains how this transfer learning method for botnet family classification was developed and how it works. In addition, this chapter contains an extension of this method. The extended version is denoted Class Balanced SBIT (or CB-SBIT for short) because it

ensures the dataset resulting after instance transfer does not contain class imbalance. Class imbalance is undesired because it can lead to overfitting among other problems in machine learning.

4.1 Introduction

Botnet Detection has been an active research area over the last few decades. Researchers have been working hard to develop effective techniques to detect botnets. From reviewing existing approaches, it can be noticed that many of them target specific botnets and many others try to identify any botnet activity by analysing network traffic. They achieve this by concatenating existing botnet datasets to obtain larger datasets, building predictive models, and then employing these models to predict whether unseen network traffic is safe or harmful. Examples of previous works where concatenated datasets have been used include the works in (Zhao et al., 2013), (Stevanovic and Pedersen, 2013) and (Stevanovic and Pedersen, 2014).

The problem with the first approach is that data is usually scarce and costly to obtain. By using small amounts of data, the quality of predictive models will be questionable. On the other hand, the problem with the second approach is that it is not always correct to blindly concatenate datasets from different botnets. Datasets can have different distributions which means they can downgrade the predictive performance of machine learning models. The approach proposed in this chapter is a novel transfer learning approach that utilises datasets from different but related domains. The idea is instead of concatenating datasets, transfer learning can be used to carefully decide what data to use. ***The hypothesis is that predictive performance can be improved by using transfer learning across datasets containing network traffic from***

different botnets. The novel approach presented in this thesis is compared to a classical open source transfer learning algorithm called TransferBoost. Experiments show that the proposed method outperforms the TransferBoost approach and produces higher accuracy. Not only this, but it is also faster which gives it another advantage. The dangers of botnets are becoming more widespread; an example of this is the WannaCry attack that caused many significant institutions in several countries to struggle to perform their services (see Section 2.1.2.4). The research community has been actively trying to develop automatic techniques to identify botnets in order to stop their harmful activities. SBIT and CB-SBIT methods can be used to enhance the performance of predictive models to identify 'mormal' and 'malicious' traffic. The subsequent section presents TransferBoost, SBIT and then CB-SBIT. Extensive comparative analysis is presented Chapter 5.

This chapter contains the following key contributions: 1) It contains a summary and overview of two existing commonly used open source transfer learning and data sampling algorithms (namely TransferBoost and SMOTE respectively), 2) It presents and explains what is meant by instance (or in general, vector) similarity and how it can be measured, 3) It introduces the novel transfer learning algorithm and highlights its strength and weakness and 4) It proposes an extension of this novel algorithm to overcome its weakness.

4.2 Methods

The following subsections provide an overview of the open source transfer learning algorithm that was used.

4.2.1 The TransferBoost Algorithm

The TransferBoost algorithm (Eaton and desJardins, 2011) is a transfer learning algorithm that is based on the classical AdaBoost algorithm. It is an instance transfer learning algorithm and it works by trying to boost target data by transferring instances from the source data and assigning weights to these instances. It examines the transferrability of instances by checking the change in performance on the target task when, and when not, transferring instances. The weight assignment in TransferBoost is done in such a way that higher weights are assigned to instances that show positive transferrability and lower weights are assigned to instances that show negative transferrability. It iteratively updates weights so that, when it finishes training, instances that exhibit positive transfer can have high weights, and therefore they have more influence, and instances that exhibit negative transfer can have very low, or zero, weights, and therefore they have little to no influence.

The TransferBoost's algorithm developer has made the implementation publicly available. The implementation is in Java and it is based on WEKA. It was downloaded and used in the experiments as will be explained in more detail in Section 5.4.

As shown in Figure 4.1 TransferBoost concatenates all the source datasets with the target datasets and assigns initial weights to instances of the newly created dataset. It then creates an initial model using this dataset and it computes a weight for this model and new weights for the instances according to their transferrability. After this it creates another model for which it computes a corresponding weight and uses this model to assign new weights to instances. This is repeated k times and in the end k models are created The value of k is a TransferBoost parameter that can be predefined. Its default

value is 10. . A prediction for an unseen instance is done by finding a weighted majority vote of the predictions of all the created k models.

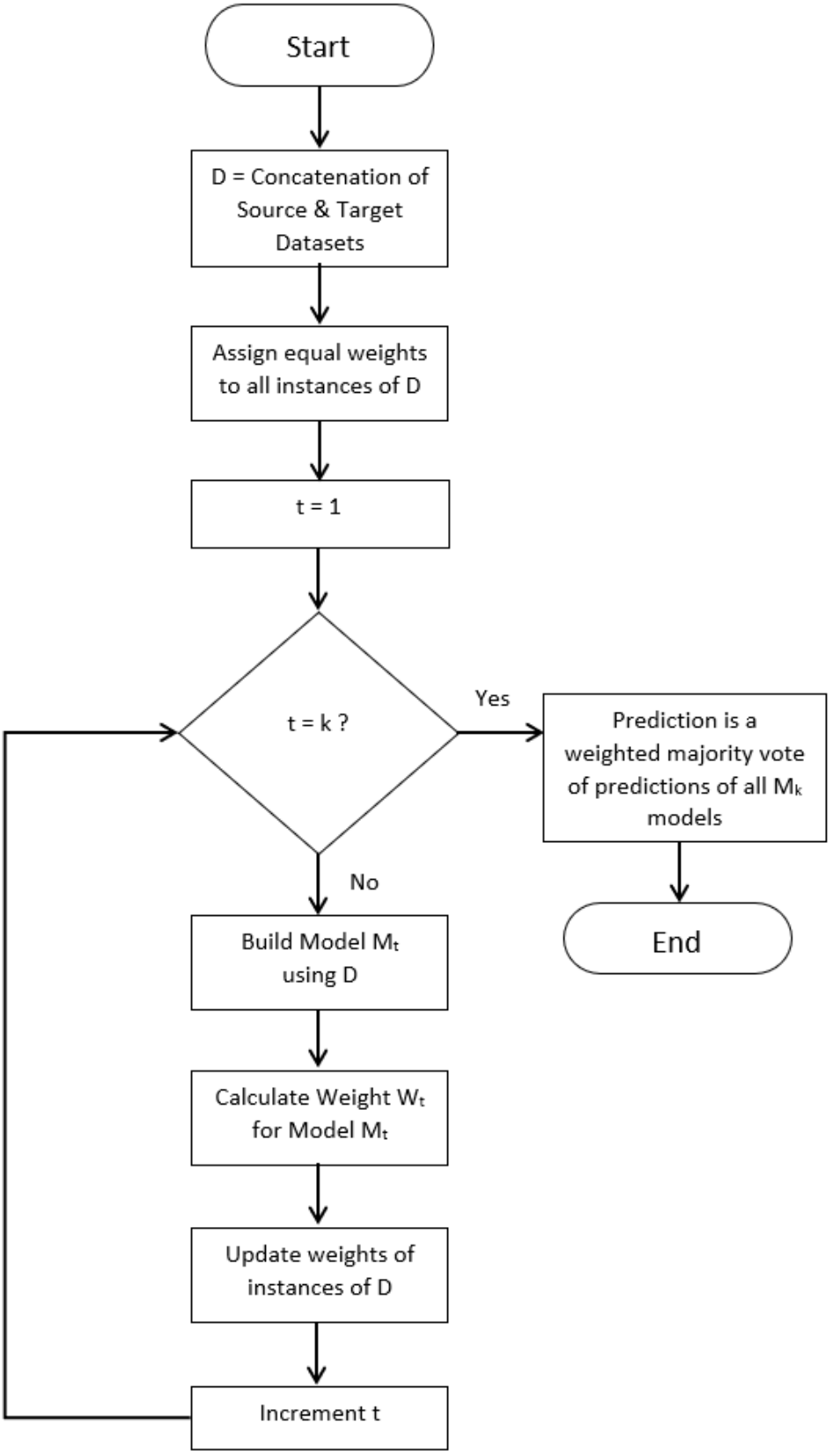


FIGURE 4.1: A Flowchart of the TransferBoost Algorithm

4.2.2 The Similarity-Based Instance Transfer (SBIT) Algorithm

In this section the proposed transfer learning method is going to be explained. Most of its details were published in (Alothman, 2018*b*) by the author of this thesis. As it was mentioned previously, the method is based on instance transfer. The algorithm receives as input one target dataset and one or more source datasets. As shown in Figure 4.2, it loops through the instances of each source dataset and checks how similar these instances are to the instances of the target dataset.

For example, imagine a situation where one wants to develop a machine learning model to accurately detect network traffic generated by botnet X and there is only little amount of labelled data that can be used (let us refer to data that belongs to class X as D_X). Using just this data to build an accurate predictive model might not be possible due to the small size of the data. Imagine there are two large labelled network traffic datasets that belong to two different botnets Y and Z respectively (let us refer to these two datasets as D_Y and D_Z). How can datasets D_Y and D_Z be utilised to improve the quality of the model developed using dataset D_X?. Concatenating D_X, D_Y and D_Z to generate a large dataset might not be the correct course of action because the datasets can have different distributions and this can lead to a poorer model. The approaches proposed and discussed in this chapter are based on carefully selecting instances from the datasets D_Y and D_Z and appending these instances to the dataset D_X. The selection process is based on the degree of similarity between instances in D_X and instances in D_Y and D_Z.

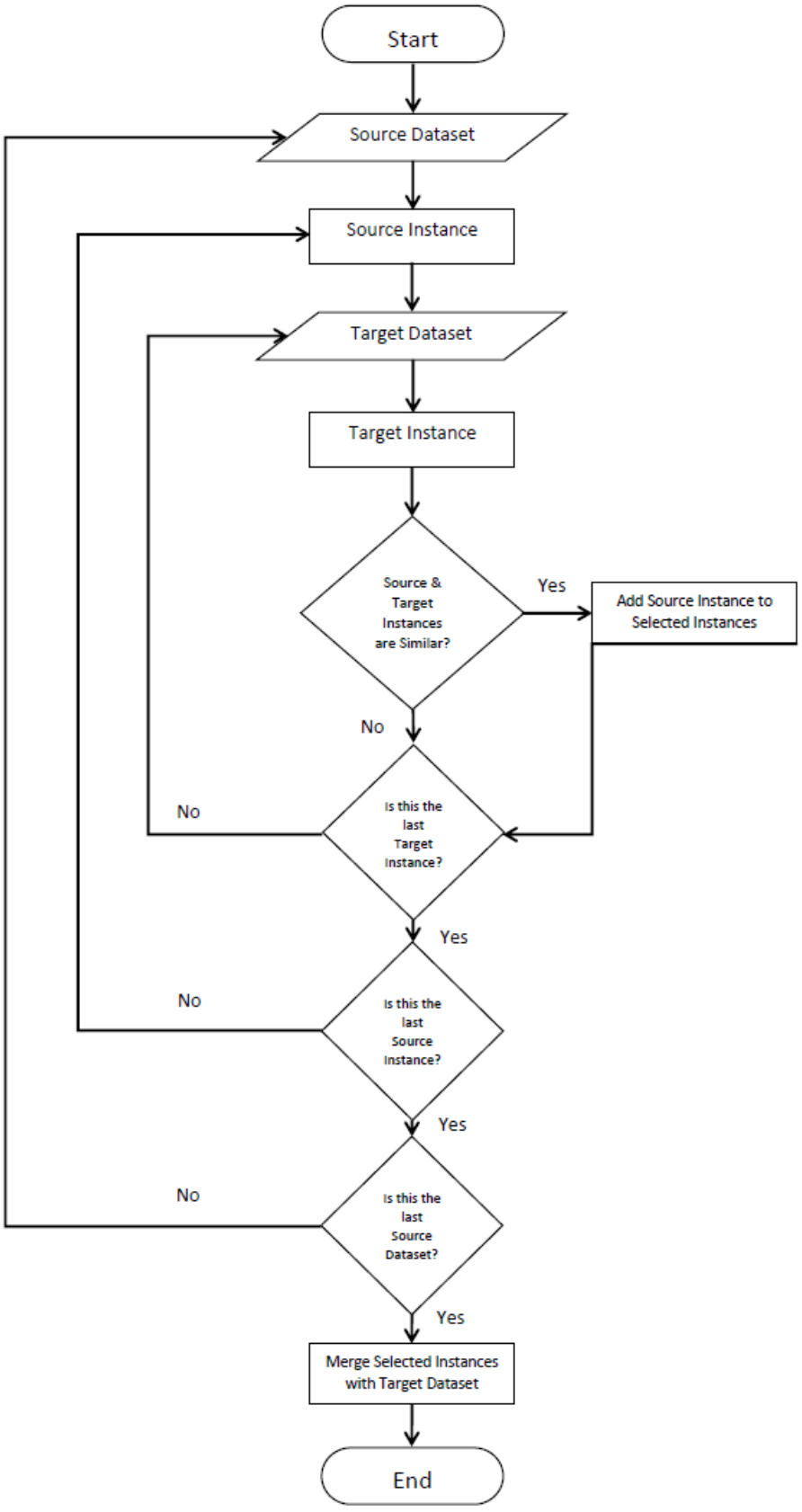


FIGURE 4.2: A Flowchart of the SBIT Algorithm

Algorithm 1 shows the pseudo-code of the SBIT approach. As the data is numerical, As many as five similarity methods are used to check how similar the instances are. An instance is selected for transfer from the source dataset to the target dataset if it satisfies the conditions. These conditions are based on using empirically determined threshold values for each type of similarity that was used. In more detail, several experiments were carried out using various threshold values and the ones that lead to improved performance were selected. The idea is, because the source datasets are related to the target dataset, they are likely to contain similar instances (similar and not necessarily identical). In other words, botnets have a similar architecture and communication mechanism. This means the data they send and receive can be similar.

Algorithm 1: The Proposed Transfer Learning Method Algorithm: Similarity-Based Instance Transfer (SBIT)

Input : Source Datasets $S_1, S_2, \ldots S_n$

Input : Target Dataset T

Input : Selected = [ ]

Input : $thr_1, thr_2, \ldots thr_k$

Output: New Dataset that is the result of $Merge(T, Selected)$

```
1  for S ∈ [S_1, S_2 ... S_n] do
2      for I_s ∈ S do
3          for I_T ∈ T do
4              Sim_1 = ComputeSimilarity1(I_s, I_T);
5              Sim_2 = ComputeSimilarity2(I_s, I_T);
6              ...;
7              Sim_k = ComputeSimilarity_k(I_s, I_T);
8              if Sim_1 > thr_1 & Sim_2 > thr_2 ... & Sim_k > thr_k then
9                  Add I_s to Selected ;
10 T_NEW = Merge(T, Selected);
11 Return T_NEW;
```

In more detail, the source datasets are scanned one by one and an attempt is made to find any similar instances in these datasets to any of the instances of the target dataset. In the remaining parts of this chapter, the developed method will be referred to as SBIT (short for Similarity-Based Instance Transfer).

The definition of similarity and how it can be calculated is going to be explained in Section 4.3. Observe that lines 4, 5 and so on in Algorithm 1 do not mention what the

type of calculated similarity is, nor do they specify how many similarity types should be calculated. This is left to the user and it can be modified to suit the application area where this algorithm is being used. It must be said that the current version of this work requires manual specification of the similarity types used as well as the thresholds that are used to decide whether two instances are similar or not (a different threshold can be used for each similarity type). The check is performed in line 8 in Algorithm 1 and, as it illustrates, one or more similarity thresholds must be exceeded for an instance from a source dataset to be deemed similar to an instance from the target dataset (and hence it is marked for copying to the target dataset). The currently used values for the similarity threshold are manually set and an interesting extension to this algorithm would be to determine these values automatically (perhaps by using a search approach such as genetic algorithms).

4.3 Instance Similarity

Before discussing the types of similarities that were used, it is logical to explain why instance similarity was used. The idea is that similar instances tend to belong to similar classes (i.e. they have similar behaviour). This is believed to be a reasonable rule-of-thumb in the absence of more detailed knowledge. Also, this is going to be examined experimentally and its worthiness will be investigated (see Section 5.4 for further details).

4.3.1 What is Similarity

The definition of similarity can be subjective; therefore, it is essential to have a quantitative approach for estimating the degree of resemblance between two or more

entities. When one says entities A and B are similar, it is important to check how similar they are, or in other words, in what aspect(s) they are similar. For example, one might claim that a circle and a triangle are similar. Although several differences such as the angles in a triangle and the fact that a circle has an infinite number of lines of symmetry can be immediately listed, one aspect of similarity is that both are closed geometric shapes.

Let us assume that the similarity of two entities is measured as a real number S. Therefore, it is common to make sure that the value of S is:

$$0 <= S <= 1$$

This is interpreted as a value of $S = 0$ means there is no similarity at all between the two entities, whereas a value of $S = 1$ means the two entities are identical (i.e. They are the same). The degree of similarity increases as S approaches 1, and decreases as S approaches 0.

4.3.2 How to Measure the Similarity of Instances

To measure the similarity of two instances, one approach is to consider the feature values of each instance as a vector (only feature values without the class label). Fortunately, the feature values in the network traffic data used in this work are all numeric.

For example, a feature vector representing one instance should look like:

$$[f_1, f_2, \ldots f_n]$$

Where $f_1, f_2, \ldots f_n$ are the feature values for the first feature, second feature and so on. There are many different formulae for computing the similarity between two numerical

vectors of the same length. The reader is referred to the book in (Deza and Deza, 2009) for an explanation of various similarity measures.

4.3.3 The Similarity Types used in this Work

Now let us assume there are two real-value vectors X and Y such that:

$$X = [x_1, x_2, \ldots x_n]$$

and

$$Y = [y_1, y_2, \ldots y_n]$$

To compute similarities between X and Y, one only needs to plug these vectors in a suitable similarity formula (Warrens, 2016). The similarity types used in this work are listed in Table 4.1.

For example, imagine there are two instances I_{zeus} and I_{sogou} that belong to the two botnets *Zeus* and *Sogou* respectively, and these two instances have the same feature space (i.e. the name and number of features is the same in both). In order to compute the similarity between I_{zeus} and I_{sogou}, their feature values can be used in one of the formulae (see Table 4.1) and the result can be easily obtained. Although it has been mentioned previously, it is important to bear in mind that different similarity types look at different similarity aspects. Thus, sometimes it is a good idea to use more than one type of similarity.

Similarity	**Formula**
Tanimoto Similarity (X,Y)	$\frac{\sum_{i=1}^{n}(x_i,y_i)}{\sum_{i=1}^{n}x_i^2+\sum_{i=1}^{n}y_i^2+\sum_{i=1}^{n}(x_i,y_i)}$
Ellenberg Similarity (X,Y)	$\frac{\sum_{i=1}^{n}(x_i+y_i)1_{x_i\cdot y_i\neq 0}}{\sum_{i=1}^{n}(x_i+y_i)(1+1_{x_i\cdot y_i=0})}$, $1_{x_i\cdot y_i\neq 0}=\begin{cases}1, & \text{if } x_i\cdot y_i\neq 0\\ 0, & \text{otherwise}\end{cases}$
Gleason Similarity (X,Y)	$\frac{\sum_{i=1}^{n}(x_i+y_i)1_{x_i\cdot y_i\neq 0}}{\sum_{i=1}^{n}(x_i+y_i)}$, $1_{x_i\cdot y_i\neq 0}=\begin{cases}1, & \text{if } x_i\cdot y_i\neq 0\\ 0, & \text{otherwise}\end{cases}$
Ruzicka Similarity (X,Y)	$1-\frac{\sum_{i=1}^{n}min\{x_i,y_i\}}{\sum_{i=1}^{n}max\{x_i,y_i\}}$
BrayCurtis Similarity (X,Y)	$\frac{2}{n(\overline{x}+\overline{y})}\sum_{i=1}^{n}min\{x_i,y_i\}$

TABLE 4.1: Different Similarity Measure Types and their Formulae

The following is a summary of how each similarity value in Table 4.1 is calculated. Observe that this is based on the assumption that the two real-value input vectors are of the same length (i.e. they contain the same number of elements). If the two vectors have different lengths, their lengths should be made equal through up/down sampling before measuring their similarity.

- **Tanimoto:** The Tanimoto similarity of two real-value vectors is calculated as a fraction of the following structure: the numerator only contains the dot product of the two input real-value vectors, whereas the denominator contains the sum of (1) the sum of the squared elements of the first input vector (2) the sum of the squared elements of the second input vector (3) the dot product of the two input vectors

- **Ellenberg:** The Ellenberg similarity is calculated as a fraction of sums of corresponding elements in the two input real-value vectors. The sum in the numerator is calculated as follows: if any of the corresponding elements is zero, then the value of their sum is zero, otherwise the correct sum is used. As for the denominator, if any of the corresponding elements is zero, then the value of their sum is the non-zero element, otherwise the correct sum is multiplied by two and used.

- **Gleason:** The Gleason similarity is similar to Ellenberg similarity explained in the previous point. The numerator is calculated in the same way. The denominator is calculated as the sum of corresponding elements in the two input real-value vectors.

- **Ruzicka:** To calculate Ruzicka similarity, firstly, the following sums are computed: (1) the sum of the minimum of corresponding elements in the two input vectors (2) the sum of the maximum of corresponding elements in the two input vectors. Secondly, the sum obtained in (1) is divided by the sum obtained in (2) and the result is subtracted from 1.

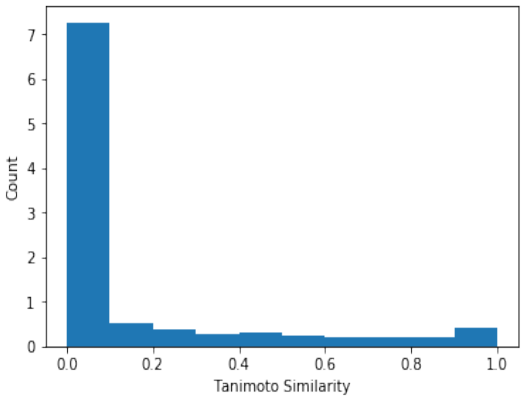
- **BrayCurtis:** The BrayCurtis similarity requires finding the average of each of the two input real-value vectors, summing these averages and multiplying the result by the number of elements in one of the input vectors. Then two is divided by the result and the resulting value is multiplied by the sum of the minimum of corresponding elements in the two input vectors.

4.3.4 Example Similarity Values

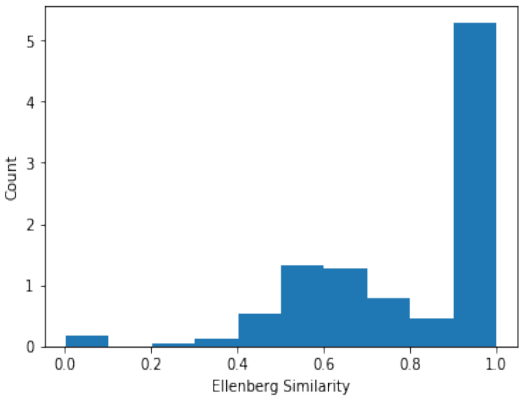
This section illustrates similarity values between instances of the *Zeus* and *Menti* botnets. As the histograms in Figure 4.3, the five similarity measures explained in Section 4.3.3 were computed between all instances and the histogram were plotted to show the distribution of similarity values.

It can be immediately noticed that the Tanimoto, Ruzicka and BrayCurtis similarities are skewed towards lower values (Figures 4.3a, 4.3d and 4.3e respectively), whereas Ellenberg and Gleason similarities are skewed towards higher values (Figures 4.3b and 4.3c respectively). This means that there is in general low similarity between *Zeus* and *Menti* data when using Tanimoto, Ruzicka and BrayCurtis similarities. On the other hand, there is in general high similarity between *Zeus* and *Menti* data when using Ellenberg and Gleason similarities. This could be attributed to the fact that different similarity measures compute similarity based on different aspects. For example, one type of similarity may focus on the port numbers and used protocols, and another type of similarity may focus on some other features. In fact, this is the main reason why multiple similarity measures were used.

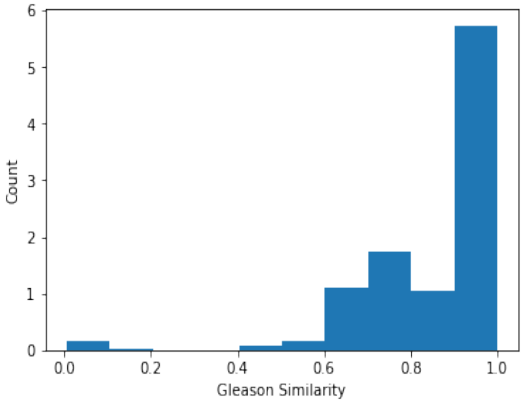
It is possible to compute the similarity between all instances in source and target datasets and to use the mean, or median, of the resulting values as the threshold for each similarity type separately. For example, an experiment as the one shown in Figure 4.3 can be run prior to SBIT and the mean of each similarity type can be used as a threshold for that particular threshold for that particular similarity.



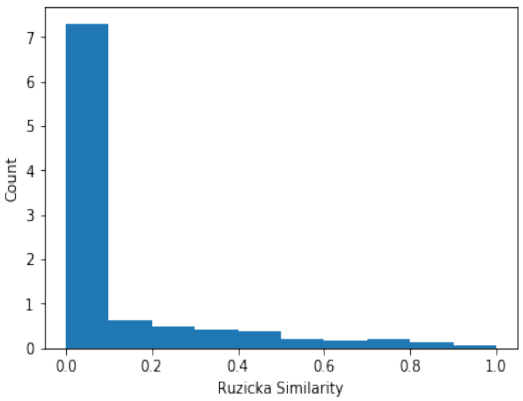
(A) Tanimoto Similarity



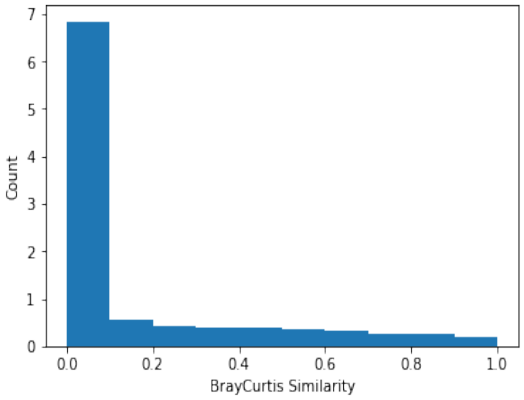
(B) Ellenberg Similarity



(C) Gleason Similarity



(D) Ruzicka Similarity



(E) BrayCurtis Similarity

FIGURE 4.3: Histogram of Similarity Values

4.4 SBIT Limitations and Extension

Careful inspection of Algorithm 1 reveals that SBIT copies an instance from the source data to the target data as soon as it satisfies the similarity criteria (lines 8 and 9). It performs this step without paying attention to the class of that instance. This means it is possible for instances transferred by SBIT to belong to one class only (or at least for the majority of them to belong to the same class) which leads to creating a new target dataset that is class imbalanced.

4.4.1 The Class Imbalance Problem

One of the main reasons that cause overfitting (Section 4.4.2) is class imbalance (He and Ma, 2013). Class imbalance refers to the problem when a classification dataset contains more than one class and number of instances in each class is not approximately the same. For example, there might be a two-class classification dataset that contains 100 instances where the number of instances for one of the classes is 90 and for the other is 10. This dataset is said to be *imbalanced* as the ratio of first class to second class instances is 90:10 (or 9:1). One might train a model that yields 90% accuracy but in reality it could be that the model is predicting the same class for the vast majority of testing data.

There are several ways to combat class imbalance (Chawla, 2010). One of these methods is to down sample the majority class (this is sometimes referred to as under sampling). In other words, to randomly select a subset of the instances that belong to the majority class so that the number of instances in each class in the resulting dataset is approximately the same. Another method is to over sample the minority class; which means to randomly duplicate instances from the minority class so the dataset becomes class balanced.

One common technique that falls under this category is the SMOTE algorithm (or the Synthetic Minority Over-sampling Technique (Chawla et al., 2002)) which generates synthetic instances that belong to the minority class rather than generating duplicates. See Section 4.4.3 for more details on how SMOTE works.

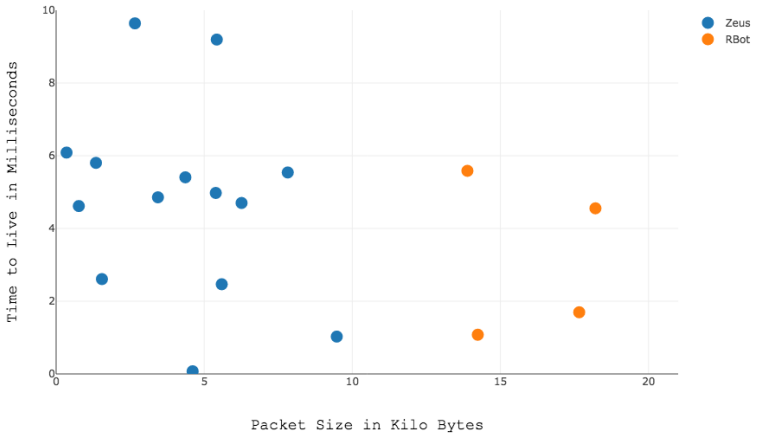
4.4.2 What is Overfitting?

Overfitting and underfitting are two of the most common challenges in machine learning (Bramer, 2013). Overfitting happens when a model fits the data it is trained on too well. It occurs when a model, not only learns the details in training data, but also the detail in the noise to the extent that it negatively affects the performance of the model on unseen data (Simonson, 2013). What this means is that the model picks up the noisy patterns in the training data which might not necessarily exist in new data. As a result, the model's ability to generalise is negatively affected. Underfitting on the other hand refers to the phenomenon when a model poorly models the training data and fails to generalise to unseen data. It can be identified early during the model training process by observing the performance of the model on the training data (it will be poor).

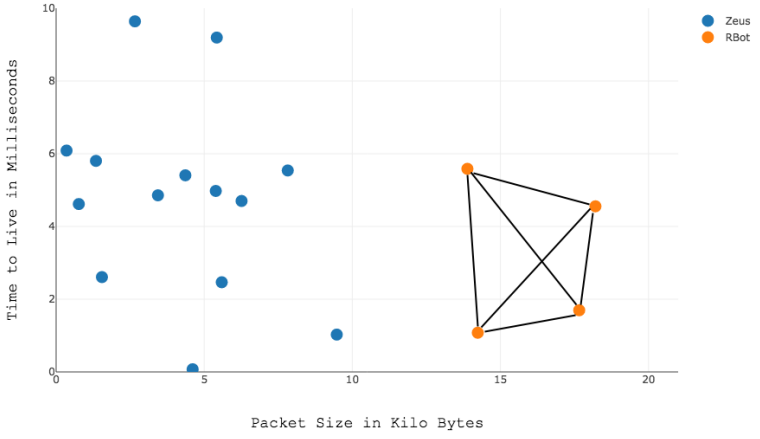
4.4.3 The Synthetic Minority Over-sampling Technique (SMOTE) Algorithm

SMOTE, or Synthetic Minority Oversampling Technique (Chawla et al., 2002), is a statistical approach that increases the number of instances in a dataset so that the dataset is class balanced. In other words, the technique works by creating new instances from already existing instances. These already existing instances are usually the minority

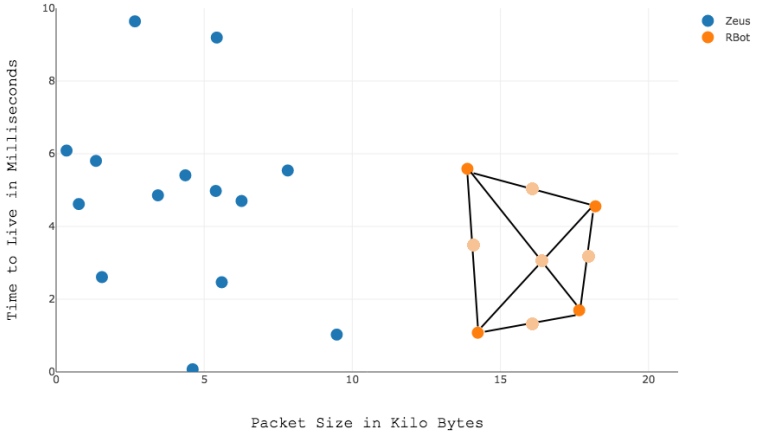
cases that are supplied as input to the algorithm. Observe that SMOTE normally does not alter the number of instances of a majority class.



(A) Original Instances



(B) SMOTE Works on Minority Instances



(C) SMOTE Creates New Instances from Original Instances

FIGURE 4.4: How SMOTE Works

The new instances generated by SMOTE are not just duplicates of existing instances; instead, SMOTE creates these new instance between existing (real) instances of the minority class. In other words, the new instances are sythesised as a combination of the pre-existing real instances. SMOTE exploits the nearest neighbours as shown in Figure 4.4.

Figure 4.4 shows a scatter plot of instances from *Zeus* and *RBot* botnets using two features only. As depicted, the way SMOTE works can be explained as follows: SMOTE draws imaginary lines between nearest instances (Figure 4.4b) and creates the required number of instances on those lines (Figure 4.4c). This makes the new synthetic instances more general (i.e. they have the same distribution as the original instances).

One issue that can be noticed about how SMOTE works is the number of neighbours. If the algorithm is given as parameter the value 1 as the number of neighbours, or there is only one instance of the minority class, then it does not work. If it creates any new instances in this case they will all be duplicates or copies of that single instance. This will be made clearer later in Section 5.4.5.

4.4.4 The Class Balanced SBIT Algorithm (CB-SBIT)

To avoid class imbalance the SBIT (Alothman, 2018*b*) algorithm discussed in Section 4.2.2 can be modified to ensure the resulting dataset is class balanced. Details of this extension were published by the author of this thesis in (Alothman et al., 2018).

Recall SBIT assumes that the target dataset is class balanced, the modified version of SBIT makes sure that the new dataset (resulting after selecting instances from source datasets) remains class balanced by using a strict criteria as illustrated in Algorithm 2. This can be achieved in more than one way. For example, it can be done on the fly

by keeping track of the ratio of classes of instances transferred from the source datasets and ensuring that whenever an instance is added, the ratio remains almost the same. In other words, it guarantees that approximately the same number of instances from different classes is transferred to the target dataset. Another method is to perform a post-processing step and sub-sample the instances selected for transfer in such a way that the classes are balanced. In the current implementation, both methods are available although only the latter is included in Algorithm 2 (lines 10 and 11).

Algorithm 2: Class Balanced Similarity-Based Instance Transfer (CB-SBIT)

Input : Source Datasets $S_1, S_2, \ldots S_n$

Input : Target Dataset T

Input : Selected = [ ]

Input : $thr_1, thr_2, \ldots thr_k$

Output: New Dataset that is the result of $Merge(T, Selected)$

```
1  for S ∈ [S_1, S_2 ... S_n] do
2      for I_s ∈ S do
3          for I_T ∈ T do
4              Sim_1 = ComputeSimilarity1(I_s, I_T);
5              Sim_2 = ComputeSimilarity2(I_s, I_T);
6              ...;
7              Sim_k = ComputeSimilarity_k(I_s, I_T);
8              if Sim_1 > thr_1 & Sim_2 > thr_2 ... & Sim_k > thr_k then
9                  Add I_s to Selected ;
10 ClassBalancedSelected = SubSample(Selected);
11 T_NEW = Merge(T, ClassBalancedSelected);
12 Return T_NEW;
```

The *SubSample* function in Algorithm 2 counts the number of instances in each class in the input dataset and randomly removes instances from the majority class(s) until the dataset is class balanced. As will be shown in Section 5.4.4, CB-SBIT in general improves the accuracy of SBIT due to its ability to ensure class balance.

4.5 Summary

This chapter has introduced the algorithms developed as part of this thesis (i.e. SBIT and CB-SBIT) as well as two commonly used open source algorithms (i.e. TransferBoost and SMOTE). As for the developed algorithms, the chapter included a detailed explanation of the intuition behind them, their pseudo-code and discussion. The chapter also included a detailed explanation of what similarity is, the similarity measures used in this work and how they were used in the developed novel algorithms. On the other hand, the two open source algorithms were briefly explained because they were used for evaluation purposes. The next chapter contains a detailed experimental evaluation of all these algorithms.

Chapter 5

Preprocessing of Raw Network Traffic Data and Performance Evaluation of the Proposed Methods

One of the most important aspects in machine learning in general, and in botnet detection specifically, is data. Without data no practical experiments can be conducted and therefore no concrete conclusions can be drawn. This chapter provides a detailed overview of how data used in this work was obtained and preprocessed. It also presents an extensive evaluation of the novel techniques explained in Chapter 4. Each data preprocessing step is explained in such a way so that the reader can follow what was carried out and why it was carried out. After explaining the steps, the chapter shows the results of applying these steps to the downloaded dataset. After that, the last section of

this chapter demonstrates the experimental results of using this data with the proposed approaches.

5.1 Introduction

Data preprocessing is an essential step in data-driven approaches; this is because data in the real world is often incomplete, or unsuitable, for being used by software algorithms such as those used in data mining and machine learning. For instance, data can contain missing values or unnecessary and uninformative features. Also, noise can exist in the data in the form of outliers or errors and these can influence not only models created using the data but also the interpretation of these models. Because of these issues and others, it is important to ensure that data is in an acceptable condition to be used in tasks such as automatic prediction and analysis.

Over the last few years, the cost of malicious attacks has risen to tens of millions of Pounds from UK bank accounts (Agency, 2017). However, in order to test and evaluate malicious traffic detection techniques, network traffic data is required. Obtaining Network traffic is normally captured using tools such as Wireshark (Orebaugh et al., 2007). The data is usually in raw PCAP (Packet CAPture) format which is not processable by popular data mining platforms such as WEKA (Hall et al., 2009) or Scikit-learn (Pedregosa et al., 2011). Hence, it is necessary to prepare the data and transform it into a suitable format.

An existing approach that uses Wireshark to transform PCAP data into textual data is the work in (Fowler and Hammel, 2014). While Wireshark extracts some features, it does not extract statistical metrics such as the ones generated by the open source tool that has been used in this work (more in Section 5.2.2). Although the review in (Davis

and Clark, 2011) attempts to summarise several existing preprocessing techniques, it focuses more on features extracted from traffic data. This paper (to the best of this thesis's author's knowledge) is the first that provides a detailed step-by-step explanation of various preprocessing phases.

This chapter has three main contributions: 1) It provides several steps that should be considered when carrying out network traffic data transformation from raw to a textual format, 2) It demonstrates these steps by applying them to a real, rather than simulated, data, and 3) It presents detailed performance evaluation of our novel tranfer learning algorithms presented in Chapter 4 using the datasets resulting from the network traffic data transformation.

5.2 Preprocessing Raw Network Traffic Data

The most common raw format of network traffic data is the PCAP format which is not supported by many widely used machine learning and data mining tools and platforms. Hence, as shown in Figure 5.1, it is necessary to carry out several steps to prepare the data for processing. These steps are explained in detail in the following subsections.

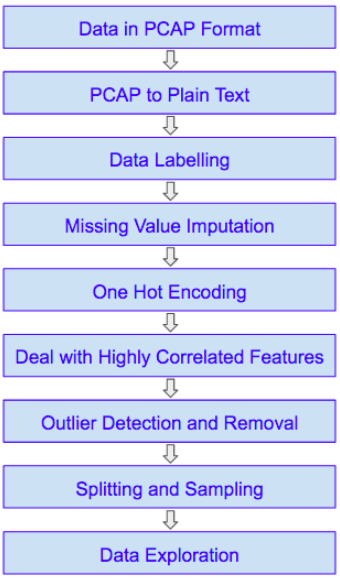


FIGURE 5.1: PreProcessingPipeline

5.2.1 Obtaining the PCAP Data

The data is usually captured in PCAP format using network traffic analysers such as Wireshark. According to the documentation of Wireshark, some global information is stored in the header of each PCAP file. After that, the file contains record(s) for captured packets. These records are organised in such a way that each packet data has its own packet record as shown in Figure 5.2.

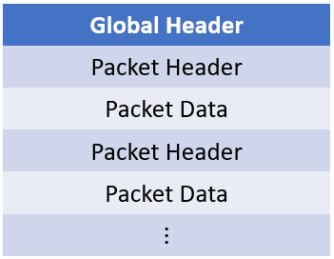


FIGURE 5.2: Contents of PCAP File

5.2.2 From PCAP to Plain Text

According to the documentation of Wireshark, network data stored in the captured packet data in a PCAP file might not necessarily be in its original order as it appeared on the network. This is because the PCAP file might store only some part of each packet (usually the length of this part is predefined to be larger than the largest possible packet so no packet is trimmed). Due to this reason, it is highly recommended to use specialised tools that understand the structure of PCAP files. Therefore, to transform PCAP data into a textual format, it is advised to use the freely available tool FlowMeter (Draper-Gil et al., 2016). This is a Java package that reads in a directory which contains one or more PCAP files and transforms them into Comma-Separated Value (CSV) files. It analyses the contents of PCAP files and generates several attributes (features) such as Source Port, Destination Port, Protocol, Flow Duration, Flow Bytes per second and Flow Packets per second. The total number of features generated by FlowMeter is 26 and their full description can be found in (Draper-Gil et al., 2016).

5.2.3 Labelling the Data:

Making predictions in data mining can be supervised or unsupervised. In supervised prediction (e.g. classification), models are trained on labelled data (data that contains features and class labels; for example, class labels for network traffic data can be Normal Traffic or Malicious Traffic). These models are then used to classify new data to predict which class it belongs to. On the other hand, in unsupervised prediction, no labels are required as the data is usually organised in clusters of relevant, or similar, instances. This step is left to the user. If classification is being carried out, then the user needs to assign labels to instances in the data. Observe that the CSV file resulting from

FlowMeter (see Subsection 5.2.2) contains a *Class* field which only has the value *ISCX*. However, if the user is applying clustering, then no advance labelling of the data being clustered is usually required; except if it was for testing or evaluation purposes.

5.2.4 Missing Value Replacement (Imputation):

One of the problems in real-world data is missing values. The existence of such phenomenon means the data is incomplete. In data analysis, there are several techniques to deal with such cases (Pigott, 2001). One of these techniques is to remove rows, or columns, that contain missing values - especially if the percentage of missing values is high. Another technique is to generate a reasonable replacement for each missing value. Generating an estimate can be done by using the feature mean for numeric values (or majority for nominal values). Another approach is to use the median instead of the mean or to use a learning algorithm such as k-Nearest Neighbor or decision tree (Rahman and Islam, 2011) to predict the missing value. Using multiple imputations, where several possible values of a missing value are generated to obtain several parallel full datasets and then combining results of analyses using these datasets, is also a common approach (ALLISON, 2000). The CSV data generated by FlowMeter can have missing values. As will be explained in Section 5.3, missing values in each column were replaced by the median of existing values of that column.

5.2.5 One Hot Encoding:

One hot encoding (Guo and Berkhahn, 2016) is a technique used to represent categorical variables as binary vectors (1 and 0). The categorical values are mapped into integer values where binary vectors are used to represent unique values (elements of each binary

vector are all zeros except the index of the categorical value, which is given the value of 1). As for the data generated by FlowMeter, this is relevant to the features *Source Port*, *Destination Port* and *Protocol*. These fields appear as integers in the generated CSV file but in reality they are categories. In data mining, it is not recommended to represent categories as numbers because this introduces order, which may not exist in the original representation. For example, Hypertext Transfer Protocol (HTTP) works on port 80 and File Transfer Protocol (FTP) works on port 21. If numerical representation is used, then data mining algorithms can assume that HTTP is larger than FTP, because 80 is larger than 21, and this is not true. Another reason to represent categorical values in the binary format using one hot encoding is that many data mining algorithms, such as neural networks (Haykin, 2007), work best with numerical data, or might not work at all with categorical data.

5.2.6 Removal of Highly Correlated Features

Having highly correlated variables (or features) in data analysis can have negative consequences when interpreting models. The existence of such variables in data can be problematic because they can influence the variance of model coefficients making them change unpredictably even to small changes in the model (Xue and Qu, 2017). Many of the features generated by FlowMeter are highly correlated. This can be understood from the explanation provided by the developers (Draper-Gil et al., 2016) and by using the tool and then generating a correlation matrix, or heatmap, for the resulting data. Since highly correlated variables can influence the performance of data mining and machine learning algorithms, it is a common practice to remove them.

5.2.7 Outlier Detection and Removal:

An outlier is an observation point (i.e. instance) that is distant from other observations (Aggarwal, 2013). The existence of outliers in the data can affect the distribution of the data; in fact, having many outliers can introduce noise into the data. Hence, it is a common practice in data analysis to detect and remove outliers. There are several techniques for outlier detection in the literature (Hodge and Austin, 2004). The process has been given several names by authors. For example, the following terms can be found: novelty detection, anomaly detection, noise detection, deviation detection or exception mining. These all refer to the same process of outlier detection that entails, given a sample, identifying the point (or points) in this sample that seem to differ noticeably from other points in this sample. Observe that in many cases in cyber security, outliers are often the instances of interest. Hence, this step is only presented here as an optional step; although applying it depends on the purpose of the application. For example, when attempting to distinguish between normal and harmful traffic, removing outliers might not be the best option, whereas, if the purpose is to distinguish between different botnet types, then removing outliers might enhance performance.

5.2.8 Splitting and Sampling:

Although it may be optional, a step that can be applied by those working in automatic identification of malicious network traffic is to create separate datasets for different classes. For example, teams might be targeting a specific botnet attack. Thus, using data that belongs to this botnet to train a learning algorithm is a high priority step. Therefore, if there exists a large dataset that contains network traffic data from various malicious applications, as well as Normal (i.e. Safe) data, it can be useful to create

smaller sub-datasets so that each sub-dataset contains data that belongs to one class only (label).

Following this, there should now be a separate dataset for each class. It is important to make sure that each sub-dataset contains Malicious and Normal network traffic data. This makes it easy to build separate models for separate attacks using data from each specific attack type and Normal data. In data mining and machine learning, this is known as having *positive* and *negative* examples in the dataset. Here a suggestion would be to randomly select non-overlapping samples from the Normal data and append them to the Malicious data. It is worth mentioning here that each new sub-dataset should contain an approximately equal number of Malicious and Normal instances to avoid class imbalance (Chawla, 2005). After finishing this step, the data should be ready for data mining and machine learning experiments.

5.2.9 Data Exploration:

Another step (which can be optional) is to perform data exploration and inspection (e.g. visualisation). If the data is high dimensional then it is not possible to visualise it and its dimensionality should be reduced prior to this step. There are many techniques that can be used to reduce dimensionality. For instance, one of the data exploratory and analysis approaches is Principal Component Analysis (Jolliffe, 1986). Principal Component Analysis, or PCA, is a technique used to identify a smaller number of uncorrelated features (i.e. attributes or variables). It performs this by linearly combining the original features and creating a new feature space. These uncorrelated features are usually known as the *principal components*. Its main objective is to explain the highest possible amount of variance with the smallest possible number of principal components. It is commonly used as a dimensionality reduction procedure as well as an exploratory procedure to

examine whether there is separation among instances that belong to different classes. PCA is an unsupervised technique because it only uses the data (without the class variable) to operate. On the other hand, another data exploration technique is Partial Least Squares (Boulesteix and Strimmer, 2007), or PLS. This is a supervised technique that uses both the data X and the class variable Y. It is used to model the covariance structures in these two matrices (i.e. X and Y) and discover the fundamental relations between them. The idea behind it is to explain the highest multidimensional variance in Y by finding the corresponding multidimensional direction in X. It is noteworthy that before applying techniques such as PCA or PLS, two important steps that are usually applied are data scaling and normalisation; more details about such transformations can be found in (van den Berg et al., 2006). As will be explained in Section 5.3, PLS was used in the experiments conducted as part of this thesis because it considers the class label in its calculations.

5.3 Applying Steps to Real Data

This section demonstrates the results of applying the steps explained in Section 5.2.1 on a publicly available PCAP dataset. This is the dataset that was mainly used in the work carried out in this thesis. All source code that demonstrates how to programmatically apply these steps is now available on Github [1]. To apply these steps, the botnet and normal network traffic data that was used in (Samani et al., 2014) was downloaded. The data is in PCAP format, and more details about it can found online [2]. The authors have made two datasets available, a training dataset and a testing dataset. The experiments were run using the testing dataset only because it contains more botnet

[1] https://github.com/alothman/RawNetworkDataPreProcessing
[2] http://www.unb.ca/cic/datasets/botnet.html

types. Because the data is in PCAP format, it was transformed into CSV format using FlowMeter as explained in Subsection 5.2.2. The total number of instances obtained after transformation was 309000. After that, the data was labelled (Subsection 5.2.3). Each row (record) in the CSV file represents a network flow over a very short time period and this record belongs to a specific class. Labelling here means to assign a class, or category, to each row (i.e. the label can be: Normal, Zeus, Neris, ... etc). The research team that published this data provided guidelines on how to assign labels. The guidelines are based on Source and Destination IP addresses so they were implemented and used. The resulting distribution of instances and classes was as shown in Table 5.1.

The next step applied was imputing missing values. This was carried out using the median of each feature to replace any values that were missing (Subsection 5.2.4). Then, the features *Source Port*, *Destination Port* and *Protocol* were replaced with the results of one hot encoding (Subsection 5.2.5). This is because FlowMeter represents them numerically whereas in reality they are categories. Its noteworthy here that the number of features after this step increased dramatically from 26 to over 60000.

This step was followed by examining the data for highly correlated features (Subsection 5.2.6). A correlation matrix was generated and features which were highly correlated were removed. The removed features were Flow_IAT_Max, Flow_IAT_Min, Fwd_IAT_Mean, Fwd_IAT_Std, Fwd_IAT_Max, Fwd_IAT_Min, Bwd_IAT_Max, Bwd_IAT_Min, Active_Max, Active_Min, Idle_Mean, Idle_Max and Idle_Min. This was consistent with the description provided in (Draper-Gil et al., 2016). Also, this did not significantly decrease the number of features because only 13 were removed and over 60000 remained.

The data is now ready for further processing, and therefore, outlier detection and removal

Label	**Original No of Instances**	**No of Instances after Outlier Removal**
Normal	149726	66131
Weasel Bot	67915	45778
Virut	42253	18170
Neris	24070	9190
Murlo	12301	9007
Menti	4887	3283
IRC	2031	4105
Zero access	1816	343
TBot	860	306
Black hole 2	443	202
Zeus	385	83
Sogou	27	81
Smoke bot	76	21
Black hole 3	103	13
RBot	80	11
IRCbot and black hole1	39	5
Weasel Botmaster	39	1
Osx_trojan	27	1

TABLE 5.1: Number of Instances in each Class

(Subsection 5.2.7) was applied using Scikit-learn's LocalOutlierFactor (LOF) (Breunig et al., 2000). The resulting distribution of instances and classes after applying this step was as shown in Table 5.1. It is interesting to see that the number of instances in some cases was as low as 1 after applying outlier detection. This is because outlier detection and removal was applied to the large dataset that contains instances from all botnets as well as normal instances. Having a large number of instances means it is likely that no transfer learning is required and standard machine learning can be used. However, as the work carried out in this thesis demonstrates, transfer learning is needed when the number of instances is low.

The final two steps in terms of data preparation were to split the resulting dataset into smaller sub-datasets according to the class variable and to create sampled sub-datasets that contain 'Botnet' and 'Normal' traffic as explained in Subsection 5.2.8.

It was mentioned in Subsection 5.2.9 that PLS was performed on the data to check

whether instances from different botnets have different distributions. The Sub-datasets of botnets: *TBot*, *Zero_access* and *Zeus* were concatenated and PLS was applied on the resulting dataset. Plotting the scores of the first two components reveals a clear separation between separate classes as shown in Figure 5.3. Please notice that each PLS component is a linear combination of the original features. This is a strong indication that instances that belong to different botnets have different distributions. Thus instead of blindly concatenating the datasets (classes), transfer learning should be utilised to select the most suitable instances that will enhance the performance of the predictive models.

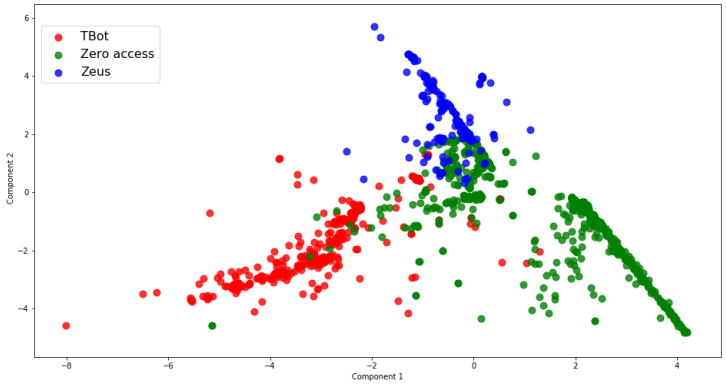


FIGURE 5.3: PLS Components 1 vs 2 for TBot, Zero_access and Zeus data

5.4 Experimental Evaluation and Discussion

In this section a detailed explanation of the experimental setups and discussion of the results is provided. The section starts with explaining how the data was prepared and then it shows an evaluation of several traditional classifiers. Then it compares the performance of the SBIT and CB-SBIT algorithms in more than one setting using mainly network traffic data but also data from the text mining domain.

5.4.1 The Network Traffic Data

To test the SBIT method and compare it to existing methods, the botnet dataset mentioned in Section 5 was used. It was split into smaller Sub-datasets according to the class label which resulted in one separate dataset for each botnet as well as one dataset that contains the Normal traffic. Non-overlapping samples from the Normal dataset were randomly drawn and added to the botnet datasets so that there are positive and negative examples in each dataset (i.e. each dataset now contains botnet and Normal data). It was ensured that the number of positive and negative examples in each dataset is approximately the same to avoid class imbalance.

As explained in Section 2.3.2, transfer learning requires Target and Source datasets. To carry out experiments, datasets which are relatively small were selected and used as Target datasets (the number of Target datasets is five). Table 5.2 shows details of each source and target dataset. Also, three datasets were selected to be used as Source datasets (this means source datasets are the same across all experiments). To evaluate performance, the Target datasets were split into two datasets: Target and Test.

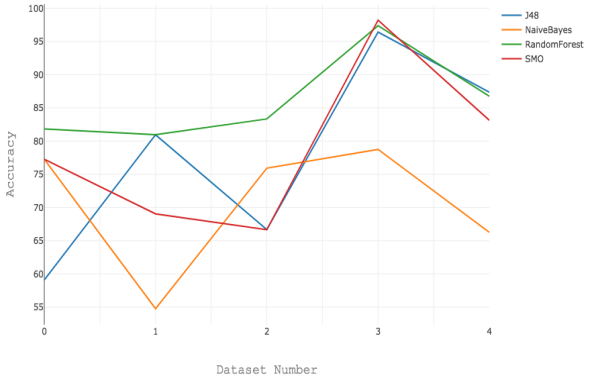
Dataset Name	**Usage**	**No of Instances**
Menti	Source Dataset	489
Mulro	Source Dataset	526
Neris	Source Dataset	501
Sogou-Target	Target Dataset	10
Sogou-Test	Evaluation Dataset	44
TBot-Target	Target Dataset	10
TBot-Test	Evaluation Dataset	231
RBot-Target	Target Dataset	10
RBot-Test	Evaluation Dataset	13
Zeus-Target	Target Dataset	10
Zeus-Test	Evaluation Dataset	165
Smoke_bot-Target	Target Dataset	10
Smoke_bot-Test	Evaluation Dataset	32

TABLE 5.2: Dataset Details

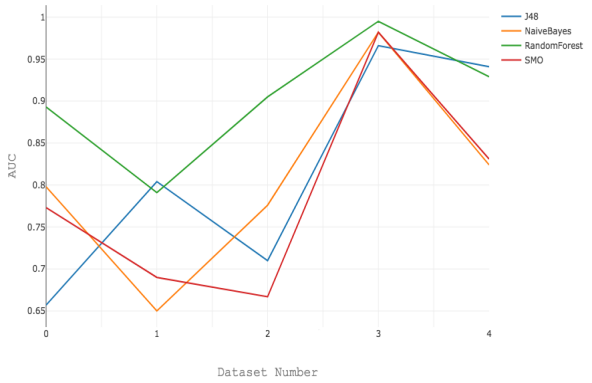
5.4.2 Evaluation of Classical Classifiers on Network Traffic Data

In this section the performance of several classical classifiers on the network traffic data that was created is presented (see Section 5.4.1 for more details about the data). Observe that these are the datasets for the five botnets: RBot, Smoke_bot, Sogou, TBot and Zeus. In the plots in Figure 5.4 these are shown in the x-axis as numbers from zero to four. Observe that these are the full datasets (i.e. before splitting them into a Target and Test datasets as shown in Table 5.2). The y-axes in the sub-figures in Figure 5.4 are the Accuracy, Area Under the Curve (AUC) and Error Rate (ERR) respectively. These

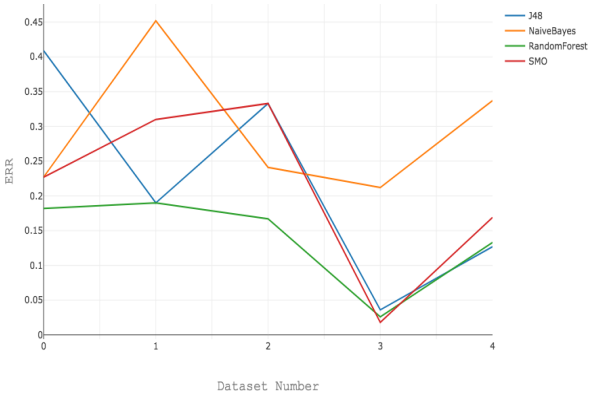
were explained in a previous chapter of this thesis (Section 3.4). The main purpose of these experiments is to select the base classifier for the transfer learning algorithm developed as part of this thesis.



(A) Accuracy



(B) Area Under the Curve (AUC)



(C) Error Rate

FIGURE 5.4: Performance of Classical Classifiers on Network Traffic Data

Figure 5.4a shows the average accuracy after running a ten-fold cross validation using

WEKA's Decision Tree (J48), NaiveBayes, RanfomForest and SMO. It can be noticed that RandomForest scored the highest accuracy in more datasets than any other classifier. In more detail, RandomForest scored the highest accuracy in four of the five datasets (i.e. datasets 0, 1, 2 and 4) and SMO scored the highest accuracy in the remaining dataset (i.e. dataset 3). All the remaining classifiers scored lower accuracies in all datasets.

In Figure 5.4b the Area Under the Curve (AUC) scored by the four classifiers. Again it is clear that RandomForest performs better than the other evaluated classifiers in most cases. It is interesting to see that the second best performer was NaiveBayes.

In order to reach a final decision, the Error Rate (ERR) was computed for the same classifiers on the five used datasets using ten-fold cross validation. Observe that in ERR usually one is interested in the minimum value (unlike accuracy and AUC where one is looking for the highest value). It is clear from Figure 5.4c that RandomForest wins again because in general it has the smallest ERR among the rest of the classifiers.

After performing the previous experiments, it becomes clear that RandomForest should be selected as the base classifier for the transfer learning algorithm developed as part of this thesis. This is because it performs better than other classifiers on network traffic data.

5.4.3 Evaluation of SBIT against RandomForest and TransferBoost

With the previous setup, comparison of the performance of the SBIT algorithm against that of RandomForest and TransferBoost was carried out. The reason RandomForest was selected is because it performs better than other classifiers on network traffic data as shown in Section 5.4.2. For RandomForest, it was trained using only the Target

datasets one at a time. For TransferBoost and the SBIT method, source and target datasets are required to perform training. The source datasets were fixed for both as mentioned previously. The Target dataset was changed using the Target datasets that were selected (i.e. the datasets that contain the word *Target* in Table 5.2). To evaluate, the accuracy of each model was computed using the corresponding test dataset. The results are illustrated in Table 5.3.

Target Dataset	**RandomForest**	**TransferBoost**	**SBIT**
RBot	83.33%	83.33%	86.58%
Smoke bot	46.87%	50.00%	63.75%
Sogou	52.27%	59.10%	77.27%
TBot	63.62%	71.42%	69.51%
Zeus	69.87%	70.51%	76.66%

TABLE 5.3: The accuracy of each Method using Different Target Datasets

Because now there are five different Target datasets (and their corresponding Test datasets), it can be seen from Table 5.3 that the SBIT method outperforms RandomForest and TransferBoost in 80% of the cases (four out of five). When experimenting with the Smoke bot data, the SBIT method produces more than 10% increase in accuracy. Also, when using Sogou data, it can be noticed that the SBIT method's accuracy is more than 17% higher than that of TransferBoost. However, TransferBoost performs better than the SBIT method when using TBot data. In addition to this, it can be seen that RandomForest was the worst performer in all experiments. It is also worth mentioning that the performance of traditional learners such as RandomForest, NaiveBayes, SVM and others was evaluated in a separate experiment. RandomForest was in general the most accurate among them and therefore

it was selected to be used it in the experiments. Another interesting comparison aspect is the run time of the SBIT algorithm against TransferBoost. Since the SBIT algorithm makes a single iteration over the Source data (as opposed to the iterative nature of TransferBoost), it is expected that the SBIT method is going to be faster. This was confirmed when run times of the previous experiments were computed as illustrated in Figure 5.5.

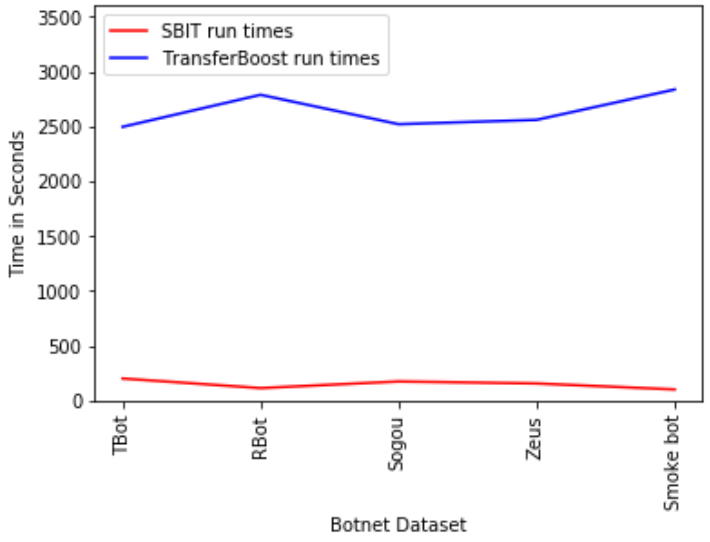
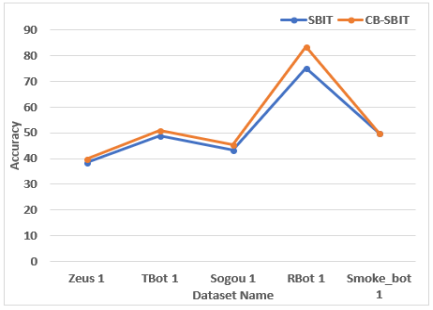


FIGURE 5.5: Run Times of the Two Algorithms

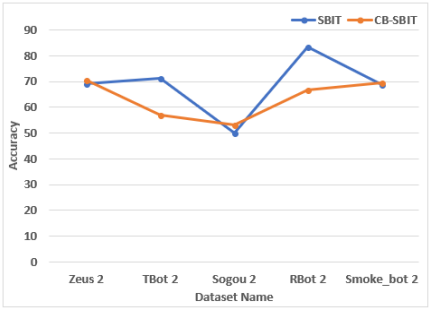
5.4.4 CB-SBIT vs SBIT

As explained in Section 4.2.2, SBIT and its extension CB-SBIT work by selecting instances from source datasets and transferring those instances to the target dataset. Currently the difference between the two algorithms is that CB-SBIT makes sure the new target dataset contains equal percentage of classes. In order to compare the two algorithms against each other, varying sizes of small network traffic datasets were created. ***The reason work was done on small datasets is that transfer learning is normally applied when data is scarce.*** These datasets are the same datasets used in (Alothman, 2018*b*) (i.e. network traffic data that belong to the following five botnets:

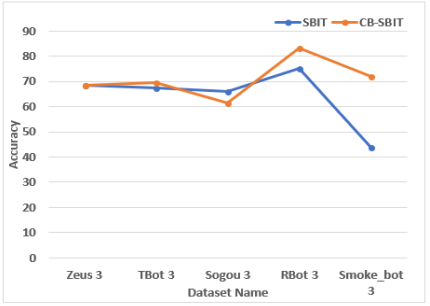
Zeus, *TBot*, *Sogou*, *RBot* and *Smoke bot*). As explained in detail in (Alothman, 2018*b*), each of these botnets has a target and testing datasets. Datasets that contain network traffic from *Menti*, *Murlo* and *Neris* botnets were used as source datasets.



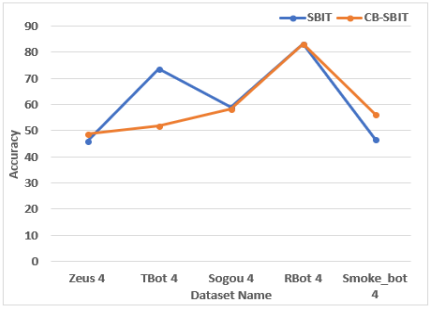
(A) Dataset 1×1



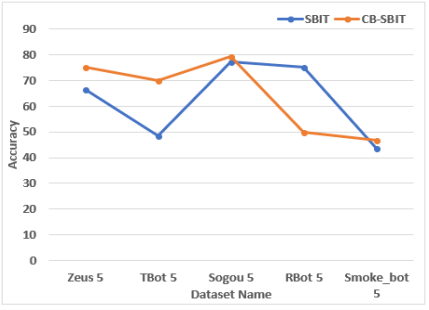
(B) Dataset 2×2



(C) Dataset 3×3



(D) Dataset 4×4



(E) Dataset 5×5

FIGURE 5.6: Accuracy Values for CB-SBIT and SBIT

The contents of these datasets are derived from the freely available raw botnet network traffic data which can be found in (Samani et al., 2014). As this dataset is in raw format, FlowMeter (Draper-Gil et al., 2016) was used to generate several features that

include statistical values as well as information such as Source Port, Destination Port and Protocol. Several steps were performed to transform this data into a suitable format for machine learning. All of these steps are explained in detail and published by the author of this thesis in (Alothman, 2018*a*).

To perform experiments, the size of each target dataset was varied in such a way that each time the target dataset contains two, four, six, eight and ten instances (it was ensured that each dataset contains the same number of botnet and normal traffic to guarantee class balance). Then SBIT and CB-SBIT were run on each of these datasets and evaluated their performance by computing the accuracy using the corresponding test dataset for each botnet. The accuracy values are illustrated in Figure 5.6. A description of the target datasets is provided in the first column in Table 5.4 in Section 5.4.5.

It is important to observe that although there are several metrics that can be used to evaluate the performance of classifiers (Japkowicz and Shah, 2011), only the accuracy was used (accuracy is the percentage of predictions that a model gets right). The reason is that the test datasets are class balanced.

Figure 5.6 illustrates the results of comparing the performance of CB-SBIT against that of SBIT using the experiment's datasets. It shows that CB-SBIT performs better than SBIT in general. Out of the 25 target datasets which were used, CB-SBIT outperforms SBIT in 16 of them. However, SBIT still outperformed CB-SBIT in 6 datasets and they performed equally on three datasets.

5.4.5 CB-SBIT vs SMOTE (using Network Traffic Data)

The way SBIT and CB-SBIT work means new real data is being added to the target dataset. Real data means the data is not synthetically generated but rather it is collected

from its original source. A common algorithm that is used to generate synthetic data is the SMOTE algorithm (or the Synthetic Minority Over-sampling Technique (Chawla et al., 2002)) which generates synthetic instances for a particular class in a dataset. This section compares and evaluates the performance of CB-SBIT and SMOTE. The datasets in Section 5.4.4 were used in this evaluation and their full description is provided in Table 5.4.

The size of each target dataset was varied so that each time the target dataset contains two, four, six, eight and ten instances - it was ensured that each dataset contains the same number of botnet and normal traffic to guarantee class balance. Then CB-SBIT was run on each of these datasets and saved the resulting target dataset - which now contains the original instances and instances added from source datasets. Using the number of instances of each class in all the resulting datasets, SMOTE was used to generate new datasets of similar sizes using the original target datasets as the base datasets.

Dataset Name (size)	**Size of Dataset generated by CB-SBIT**	**Size of Dataset generated by SMOTE**
Zeus 1 (1 × 1)	32 × 32	-
Zeus 2 (2 × 2)	106 × 106	106 × 106
Zeus 3 (3 × 3)	108 × 108	108 × 108
Zeus 4 (4 × 4)	138 × 138	138 × 138
Zeus 5 (5 × 5)	156 × 156	156 × 156
TBot 1 (1 × 1)	42 × 42	-
TBot 2 (2 × 2)	161 × 161	161 × 161
TBot 3 (3 × 3)	211 × 211	211 × 211
TBot 4 (4 × 4)	274 × 274	274 × 274
TBot 5 (5 × 5)	360 × 360	360 × 360
Sogou 1 (1 × 1)	44 × 44	-
Sogou 2 (2 × 2)	67 × 67	67 × 67
Sogou 3 (3 × 3)	147 × 147	147 × 147
Sogou 4 (4 × 4)	170 × 170	170 × 170
Sogou 5 (5 × 5)	252 × 252	252 × 252
RBot 1 (1 × 1)	17 × 17	-
RBot 2 (2 × 2)	34 × 34	34 × 34
RBot 3 (3 × 3)	38 × 38	38 × 38
RBot 4 (4 × 4)	186 × 186	186 × 186
RBot 5 (5 × 5)	212 × 212	212 × 212
Smoke bot 1 (1 × 1)	1 × 1	-
Smoke bot 2 (2 × 2)	52 × 52	52 × 52
Smoke bot 3 (3 × 3)	58 × 58	58 × 58
Smoke bot 4 (4 × 4)	77 × 77	77 × 77
Smoke bot 5 (5 × 5)	96 × 96	96 × 96

Table 5.4: Datasets Resulting after CB-SBIT and SMOTE

The first column of Table 5.4 shows the botnet name and the size of the baseline target dataset used (the 1×1 means this dataset contains only two instances, one botnet and one normal, the same concept applies for other sizes). The second column contains the size of the dataset after applying CB-SBIT using each target dataset as explained above ($number\,of\,botnet\,instances \times number\,of\,normal\,instances$). The third column contains the size of the dataset after applying SMOTE using each target dataset. Observe that the cells corresponding to target dataset of size 1×1 is empty. This is because SMOTE requires at least two instances of each class to work. Therefore, because SBIT (and CB-SBIT) works normally even when the target dataset contains only one instance of one or more classes, it is fair to conclude that CB-SBIT has a clear advantage when this is the case.

The performance of RandomForest using each one of them was evaluated. RandomForest was run on each dataset and the accuracy was computed using the corresponding test dataset for each botnet. The accuracy values are illustrated in Figure 5.7.

(A) Dataset 2×2 (B) Dataset 3×3

(C) Dataset 4×4 (D) Dataset 5×5

FIGURE 5.7: Accuracy Values for CB-SBIT and SMOTE

Inspecting Figure 5.7 reveals interesting results. Because SMOTE does not work when the number of instances for any of the classes in the data is less than two, CB-SBIT has a clear advantage in this case. Figure 5.7a shows a similar behaviour that CB-SBIT performs better when the dataset size is small but greater than two. When the dataset size is increased gradually, the performance of SMOTE improves and it can be said that it performs equally to CB-SBIT. After using the 25 datasets described in Table 5.4, CB-SBIT performs better than SMOTE in 17 cases, SMOTE performs better than CB-SBIT in 7 cases and the two of them perform equally in one case.

5.4.6 CB-SBIT vs TransferBoost (using Text Data)

For this comparison the popular 20 news groups dataset (Lang, 2007) was used to compare the performance of CB-SBIT against TransferBoost (Eaton and desJardins, 2011) and RandomForest. This dataset consists of 20,000 newsgroups posts on 20 topics where 1000 posts were collected for each topic. According to the guidelines provided in (Lang, 2007) the 20 groups can be generally categorised into the following six high level categories: computer (contains five sub-categories), miscellaneous (contains only one sub-category), recordings (contains four sub-categories), science (contains four sub-categories), talk (contains three sub-categories) and religion (contains three sub-categories). In order to perform experiments the following six datasets were selected (one from each category): *misc.forsale*, *comp.graphics*, *alt.atheism*, *sci.electronics*, *rec.autos* and *talk.politics.misc*.

In order to obtain data suitable for machine learning, techniques popular in text mining (Feldman and Sanger, 2006) were used. Text mining involves using several techniques to process (usually unstructured) textual information and generate structured data which can be used to create predictive models and/or to gain some insight into the original textual information. The structured data is usually extracted by analysing the words in the documents and deriving numerical summaries about them.

To be able to use the text documents belonging to the six categories, a dataset that has two columns was created: the text contained in each document the class of that document (which is one of the six categories). After that, the TextToWordVector filter in WEKA (Hall et al., 2009) was applied with Term Frequency and Inverse Document Frequency (TF-IDF) (Weiss, Indurkhya, Zhang and Damerau, 2004). TF-IDF is a widely used transformation in text mining where terms (or words) in a document are given

importance scores based on the frequency of their appearance across documents. A word is important and is assigned a high score if it appears multiple times in a document. However, it is assigned a low score (meaning it is less important) if it appears in several documents.

WEKA's default parameters for this filter were used except for the *number of words to keep*. This parameter is 1000 by default, it was changed to 10000. In addition to the TextToWordVector, also WEKA's NGramTokenizer was used (with NGramMinSize and NGramMaxSize set to two and three respectively). Not only this, but also Stop Words were removed using a freely available set of stop words. The resulting dataset contained as many as 10530 features and several thousand instances (belonging to the six classes).

The next step was to make sure datasets contained positive and negative examples. This was achieved by choosing one of the six categories to be the negative class (the misc.forsale data was randomly chosen). After this, the large dataset was split into smaller datasets according to class and randomly selected a subset of 194 instances from each dataset (except the misc.forsale dataset). Then samples from the misc.forsale dataset were randomly selected (without replacement) and appended to the other datasets. This was done to ensure that each dataset contains positive and negative instances. At the end of this step five datasets were obtained as follows: comp.graphics, alt.atheism, sci.electronics, rec.autos and talk.politics.misc (to clarify, the comp.graphics dataset now contains 388 instances, 194 of which are of the comp.graphics class and the remaining 194 are of the misc.forsale class, the same concept applies for the other four datasets).

Since transfer learning requires source and target datasets, two of the five datasets were randomly selected to be the source datasets (these were the rec.autos and

sci.electronics datasets). The remaining three datasets (comp.graphics, alt.atheism and talk.politics.misc) were the target datasets. Each of these three datasets was randomly split into smaller datasets (a target and testing datasets). Each target dataset contained 10 instances (five positive and five negative) and the remaining data was used as the testing datasets. Observe that it was ensured that non-overlapping subsets were randomly selected in all previous steps. Details of these datasets are provided in Table 5.5.

Dataset Name	**No of Instances**	**Dataset Usage**
rec.autos	388 (194 × 194)	Source dataset
sci.elecronics	388 (194 × 194)	Source dataset
alt.atheism_Target	10 (5 × 5)	Target dataset
alt.atheism_Test	378 (189 × 189)	Test dataset
comp.graphics_Target	10 (5 × 5)	Target dataset
comp.graphics_Test	378 (189 × 189)	Test dataset
talk.politics.misc_Target	10 (5 × 5)	Target dataset
talk.politics.misc_Test	378 (189 × 189)	Test dataset

TABLE 5.5: Text Dataset Details

With this setup experiments using RandomForest, TransferBoost and CB-SBIT were run. When using RandomForest, it was trained using only the target datasets one at a time. This is because RandomForest only requires one dataset as its input. TransferBoost and CB-SBIT require one Target dataset and one or more Source Datasets, therefore the source datasets were fixed as shown in Table 5.5 and changed the Target dataset using the Target datasets which have been selected. To evaluate, the

accuracy of each model was computed using the corresponding test dataset. The results are illustrated in Table 5.6.

Dataset Name	**CB-SBIT**	**TransferBoost**	**RandomForest**
alt.atheism	51.06%	89.68%	50.53%
comp.graphics	50.00%	78.84%	50.00%
talk.politics.misc	50.26%	87.56%	52.12%

TABLE 5.6: Results using Text Dataset

It is clear from Table 5.6 that when using textual data, TransferBoost outperforms RandomForest and CB-SBIT. This could be attributed to the nature of the data and how each algorithm works. It can be noticed that the performance of CB-SBIT and RandomForest are almost identical. This is because CB-SBIT uses RandomForest as its base learner and the fact that similarity values between instances in source and target datasets were found to be too small (when compared to the similarity values obtained when using network traffic data). Examples of this are shown in Table 5.7 where we compare the percentage of similarity which values are over 0.5 to the total number of similarity values obtained when comparing two datasets (observe that the number of similarity values obtained after using dataset 1 and dataset 2 is the product of the sizes of the two datasets). The first column of the table shows the two datasets used (for which instances five types of similarity were computed). The second to the sixth columns show the similarity types.

Datasets	**Tanimoto**	**Ellenberg**	**Gleason**	**Ruzicka**	**BrayCurtis**
Graphics - Autos	0.0093%	0.0093%	0.0193%	0.0093%	0.0193%
Politics - Electronics	0.0086%	0.0080%	0.0173%	0.0080%	0.0173%
Zeus - Sogou	12.6311%	91.2733%	97.3485%	7.9254%	14.1463%
TBot - Menti	2.9381%	85.6801%	99.8750%	2.0438%	3.0313%

TABLE 5.7: Percentage of Similarity Values that are > 0.5 using Text and Network Traffic Data

It is evident that there is much higher similarity in network traffic data than in text data. This means that CB-SBIT hardly finds any instances to transfer from the source to any of the target datasets when using text data. This is an interesting observation especially when it is compared to how CB-SBIT was able to transfer several instances when used with the network traffic data.

5.5 Summary

This chapter provided a detailed evaluation of the main work carried out as part of this thesis. It provided an explanation of several steps that can be applied when preprocessing network traffic data. Although some of these steps can be seen as optional, some others are essential in order to detect malicious network traffic with high accuracy. After explaining these steps, they were applied to an existing network traffic dataset. This is the dataset that was used to carry out the experiments conducted as part of this thesis. After that this data was used to evaluate the algorithms developed and compare their performance against other algorithms. The chapter provided an evaluation of several traditional classifiers such as RandomForest and NaiveBayes to justify why RandomForest was selected to be the base classifier for the SBIT and

CB-SBIT algorithms. Also, several tests were run to demonstrate the performance of SBIT and CB-SBIT against each other and against TransferBoost and SMOTE. These experiments were performed using network traffic data as well as text data. The results show that CB-SBIT performs better than the other algorithms on network traffic data whereas TransferBoost outperforms other algorithms on text data. In the next chapter the conclusions and research work are provided.

Chapter 6

Conclusions and Future Directions

This is the last chapter of this thesis. It provides several lessons learned after developing and experimentally evaluating the methods explained in the previous chapters. In addition, the chapter highlights how and why the work carried out is useful. It is important to observe that the remaining part of this chapter is broken down into specific sections and subsections for the conclusions, limitations and possible future work for each method. This is done in order to make it more focused and easier to understand by the reader.

6.1 Lessons learned from this project

Several lessons can be learned from an empirical project such as this. One of the major lessons is to ensure there is enough suitable data as obtaining data can be costly and time consuming. One should avoid assuming that there is plenty of freely-available data

and should try to have the data ready at an early stage of a project. Another lesson is to make sure enough computing power is available especially if the project involves analysing data of big size. Several computing bottlenecks were encountered during this project and an external powerful computer needed to be used.

Also, documenting and publishing methods and results can be useful especially at high-level conferences where experts of the field meet. This is beneficial because one can receive feedback and build on it. An important practice is to release source code and data so that other researchers can use them. This gives the carried out work more credentials as people can reproduce the results.

6.2 How and Why this work is useful

The work carried out as part of this thesis is intuitively compelling and it is primarily experimental. The main scientific methodology behind it is to implement any new ideas and test them on real data. This is useful especially with the large number of problems existing in the real world (problems that can be solved by machine learning). This work is useful for researchers and practitioners who address various research problems and use an empirical method to solve them (i.e. researchers with a hands-on and experimental mindset).

The methods presented in this work for Android app source code analysis are applicable in any other areas where a textual representation can be extracted from a non-textual representation. The core idea, which is to use the textual representation of an object as a basis to identify the category the object belongs to, remains the same.

In addition, the novel transfer learning algorithm introduced in this thesis can be useful in other areas where data is scarce. An example of these areas is medical data analysis

(such as automatic diagnosis of patients with rare medical conditions). Additionally, it can also be used in image analysis and object recognition applications where images of certain objects are too few.

6.3 Conclusions

This section provides the conclusions of every method developed, explained and experimentally evaluated in this thesis.

6.3.1 Android Botnet Detection

This thesis proposed an effective approach for the automatic detection of botnet apps. This approach is based on the analysis of the Java source code of such apps. The approach starts with Android apps as *apk* files, uses reverse engineering to obtain their Java source code and then analyses that source code. The source code analysis and mining was done using two techniques. In the first method, the Java source code is treated as if it was normal text by using Natural Language Processing (NLP) methods. And in the second approach, several statistical measures (i.e. metrics) from the source code were extracted and used as attributes. The developed approach can be considered static as it does not require the execution of the Android app itself. The idea is that as soon as an Android app is downloaded, it is reverse engineered and its Java source code is obtained and used to predict whether this app is safe or malicious. The advantage here is being *proactive*. In other words, an attempt is made to identify danger before it occurs.

As for the data resulting after extracting the source code metrics, only one dataset was created. On the other hand, several datasets were created after using NLP techniques.

This is because when converting text into word vectors the *number of words to keep* was varied and therefore multiple datasets were created. In addition, feature selection was applied to these datasets and tests were run using the original and feature selected versions of each dataset.

Several traditional classifiers were evaluated and multiple metrics were calculated to examine their performance. It was interesting to see that Random Forest was in general the best classifier and the preferred representation was to use 5000 *number of words to keep* and to apply feature selection.

6.3.2 Raw Network Traffic Data Preprocessing

Automatic detection of malicious network traffic is an important task that should be as accurate as possible. One of the main steps in carrying out this detection is to capture network traffic, prepare it for analysis and then perform the analysis. As part of thesis, several steps that should be considered when analysing network traffic data were provided, explained, and their results were illustrated using real freely available data. While some of these steps are optional, some others are required in order to transform data into a suitable format for data mining tools and platforms. After applying these steps to an existing open source PCAP dataset, the resulting data was used for extensive machine learning experiments as part of evaluating the transfer learning approaches proposed in this thesis.

6.3.3 Similarity Based Instance Transfer (SBIT)

This thesis has introduced a novel, fast yet effective and powerful method for transfer learning which was successfully used to classify botnet traffic. This method is an instance

transfer method that is based on the similarity between instances in the source data and instances in the target data. The method computes more than one similarity measure to make sure as much information as possible is captured. Experimental results show that this method outperforms, in general, a classical instance transfer learning algorithm, namely the TransferBoost algorithm. Not only this, but this method is also much faster which gives it another advantage.

6.3.4 Class-Balance Similarity Based Instance Transfer (CB-SBIT)

This thesis has introduced the novel SBIT algorithm and an extension to it. The extended version of the SBIT algorithm is aware of the percentage of classes in the resulting dataset (resulting after instance transfer) in the sense that it makes sure the classes are balanced. This helps in avoiding several problems such as overfitting and misinterpretation. The new version of the SBIT algorithm was called Class-Balanced SBIT, or CB-SBIT for short. The thesis also included extensive experimental evaluation of the CB-SBIT algorithm against the original SBIT algorithm as well as against two open source commonly used algorithms; the SMOTE and TransferBoost algorithm.

Experimental results showed that CB-SBIT outperforms SBIT in majority of the tests performed; which means CB-SBIT is an improvement over SBIT. When comparing CB-SBIT against SMOTE, several network traffic datasets of various sizes were used and it was evident that CB-SBIT outperforms SMOTE in small datasets (CB-SBIT seems to perform better than SMOTE as the dataset gets smaller). An interesting case was when the dataset contains only one instance of one or more classes. SMOTE does not work in this case whereas CB-SBIT functions normally. On the other hand, text data from the publicly available 20 news groups dataset was used to compare the performance of CB-SBIT against TransferBoost. It was interesting to discover that, despite the fact

that CB-SBIT (and subsequently SBIT) outperforms TransferBoost when using network traffic data, TransferBoost works much better than CB-SBIT on text data.

The reason why CB-SBIT exhibited poorer performance on the text data proved to be because of the extremely low similarity values between instances from different topics in the text data. Whereas, in the network data where the computations showed that higher similarity values were present, the performance was excellent. The differences in performance between the text and network datasets proves that the proposed 'similarity-based' methods worked as expected in the appropriate transfer learning scenario.

6.4 Limitations and Future Work

Here the limitations and possible extensions of every method developed, explained and experimentally evaluated in this thesis are provided.

6.4.1 Android Botnet Detection

One of the limitations of the approach proposed in Chapter 3 is that the system can be defeated by code obfuscation. This is a problem that usually affects static based approaches in general. In order to mitigate obfuscation, the source code metric approach is still valid since it does not rely on the code syntax (unlike the NLP approach).

Despite limited Android botnet data, the methods proposed and evaluated in Chapter 3 are still valid and applicable in practice. However, it would be desirable to evaluate the performance with a larger dataset in the future as this will allow techniques such as ensemble learning (with algorithms such as bagging, boosting, stacking, dagging ... etc)

to be explored. These algorithms tend to perform better with larger datasets (although it has been shown that boosting, by design, works well with small datasets).

Another idea that can be explored is to merge the two types of datasets created (i.e. the metrics and text mining data). This is illustrated in Table6.1 (where W's are weights resulting after TF-IDF and M's are metrics as explained in previous sections). After that, an investigation into whether any performance improvement is gained can be carried out.

App Name	**W1**	**W2**	**...**	**M1**	**M2**	**...**	**Class (botnet or not)**
App 1	0.069	0.034	...	2343	0.21	...	Yes
App 2	1.03	0.018	...	1983	0.43	...	No
...	...	...	...	...	...	...	...
App n	0.009	0	...	3261	0.37	...	No

TABLE 6.1: An Example Dataset Resulting After Merging Text Mining and Metrics Datasets

In addition to the above, the approach developed in this thesis can be applied to other mobile operating systems such as the iOS because existing research shows it is possible to reverse engineer their apps (Joorabchi and Mesbah, 2012).

6.4.2 SBIT and CB-SBIT

One of the current limitations of the SBIT method (and its extension CB-SBIT) is that similarity thresholds are predefined (i.e. they are manually set). In other words, these values are not automatically dynamically adjusted according to the data being analysed. Therefore, future work could explore whether an optimisation approach, such as Genetic Algorithms, can be exploited to find the optimal threshold used for

similarity. Note that however, this could slow down the SBIT/CB-SBIT algorithms. Hence, speed-performance trade-off could also be investigated.

Also, future work could explore computing the similarity between all instances in source and target datasets and to use the mean, or median, of the resulting values as the threshold for each similarity type separately to see if performance can be improved. In addition to that, the current versions of the algorithms only checks if the similarity is above a certain threshold, it does not check whether the similarity is 1 (i.e. the source and target instances are identical). This check is necessary to make sure data does not contain duplicates. Hence, it is intended to add this check in the near future.

The current implementation of SBIT and CB-SBIT uses five types of similarity as explained in previous chapters in this thesis. Namely, the currently used similarity types are Tanimoto, Ruzicka, BrayCurtis, Ellenberg and Gleason. This can be extended to include more similarity types. Not only this but to also use distance measures instead of similarity measures.

Appendix A

Code for transforming Java Source Code into Dataset for Machine Learning

```
import java.io.File;
import java.io.IOException;

import weka.core.Instances;
import weka.core.converters.ArffSaver;
import weka.core.converters.TextDirectoryLoader;
import weka.core.stemmers.LovinsStemmer;
import weka.core.stopwords.WordsFromFile;
import weka.core.tokenizers.WordTokenizer;
import weka.filters.Filter;
import weka.filters.unsupervised.attribute.Reorder;
import weka.filters.unsupervised.attribute.StringToWordVector;
//import weka.core.converters.ConverterUtils.DataSource;

public class Preprocess {

        public static void main(String[] args) {
                // convert the directory into a dataset
                TextDirectoryLoader loader = new TextDirectoryLoader();
                try {
                        //here each text file is transformed
                        //into one string in the dataset
                        //each string has a class (BotNet or Normal)
                        //the dir should contain two subdirs,
                        //one contains files with Normal code
                        //the other contains files with Botnet code
                        loader.setDirectory(new File("Path/To/SourceCodeDirs/"));
                        Instances rawData = loader.getDataSet();

                        //Make a filter
                        StringToWordVector filter = new StringToWordVector();

                        //Make a tokenizer
                        WordTokenizer wt = new WordTokenizer();
                        String delimiters =
                        " \r\t\n.,;:\'\"()?!-><#$\\%&*+/@^_=[]{}|‘~0123456789";
                        wt.setDelimiters(delimiters);
                        filter.setTokenizer(wt);
                        //Inform filter about dataset
                        filter.setInputFormat(rawData);

                        //number of words to keep
                        filter.setWordsToKeep(5000);
```

```
                        //apply TF-IDF transform
                        filter.setIDFTransform(true);
                        filter.setTFTransform(true);

                        //use stemming
                        LovinsStemmer stemmer = new LovinsStemmer();
                        filter.setStemmer(stemmer);

                        //filter.setLowerCaseTokens(true);

                        //use stopwords list to remove stop words
                        WordsFromFile stopWords = new WordsFromFile();
                        stopWords.setStopwords(new File("stopwords.txt"));
                        filter.setStopwordsHandler(stopWords);

                        //here is where we apply the filter
                        Instances dataFiltered = Filter.useFilter(rawData, filter);

                        //move class label to last index - the filter to use is Reorder
                        Reorder reorder = new Reorder();
                        reorder.setAttributeIndices("2-last,1");
                        reorder.setInputFormat(dataFiltered);
                        dataFiltered = Filter.useFilter(dataFiltered, reorder);

                        //set class index to the last attribute
                        //here we tell the dataset that the class is the
                        //last element/field/column
                        dataFiltered.setClassIndex(dataFiltered.numAttributes() - 1);

                        //save the dataset as ARFF file
                        ArffSaver saver = new ArffSaver();
                        //saver.setInstances(rawData);
                        saver.setInstances(dataFiltered);
                        saver.setFile(new File("data.arff"));
                        saver.writeBatch();
                } catch (IOException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }
                catch (Exception e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }
        }
}
```

Appendix B

Code for evaluating Performance of Classifiers

```
import java.io.IOException;
import java.util.Random;

import weka.classifiers.Classifier;
import weka.classifiers.Evaluation;
import weka.classifiers.bayes.NaiveBayes;
import weka.classifiers.lazy.IBk;
import weka.classifiers.trees.J48;
import weka.classifiers.trees.RandomForest;
import weka.classifiers.functions.SMO;
import weka.core.Instances;
import weka.core.converters.ConverterUtils.DataSource;

public class BotnetMain {
        /**
         * @param args
         */
        public static void main(String[] args) {
                try {
                        //here we open/load the arff file
                        DataSource data = new DataSource("data.arff");
                        //here we get the actual dataset
                        Instances dataset = data.getDataSet();
                        //here we tell the dataset that the class is in the
                        //last element/field
                        dataset.setClassIndex(dataset.numAttributes() - 1);
                        //here we just printout the number of features/attributes
                        System.out.println(dataset.numAttributes());

                        int seed = 1;  // the seed for randomizing the data
                        int folds = 10;
                        Random rand = new Random(seed);
                        double acc = 0.0;
                        //here we just want to print the classifier name
                        //and Accuracy value for each model
                        System.out.println("Model,Accuracy");

                        //create several models, run 10 fold
                        //cross validation and get Accuracy metric
                        J48 tree = new J48();
                        //Initialise evaluation with the dataset
                        Evaluation eval = new Evaluation(dataset);
                        //this is where we apply 10 fold cross-validation
                        eval.crossValidateModel(tree, dataset, folds, rand);
                        //get the accuracy, here we used double because
                        //we need the accurate number ex. 92.21
                        acc = eval.pctCorrect();
```

```
                        System.out.println("J48,"+acc);

                        NaiveBayes nb = new NaiveBayes();
                        //this is where we apply 10 fold cross-validation
                        eval.crossValidateModel(nb, dataset, folds, rand);
                        //get the accuracy, here we used double because
                        //we need the accurate number ex. 92.21
                        acc = eval.pctCorrect();
                        System.out.println("NaiveBayes,"+acc);

                        IBk knn = new IBk(5);
                        //this is where we apply 10 fold cross-validation
                        eval.crossValidateModel(knn, dataset, folds, rand);
                        //get the accuracy, here we used double because
                        //we need the accurate number ex. 92.21
                        acc = eval.pctCorrect();
                        System.out.println("KNN,"+acc);

                        RandomForest rf = new RandomForest();
                        //this is where we apply 10 fold cross-validation
                        eval.crossValidateModel(rf, dataset, folds, rand);
                        //get the accuracy, here we used double because
                        //we need the accurate number ex. 92.21
                        acc = eval.pctCorrect();
                        System.out.println("RandomForest,"+acc);

                        SMO smo = new SMO(); //SMO(Sequential Minimal Optimisation)
                        //this is where we apply 10 fold cross-validation
                        eval.crossValidateModel(smo, dataset, folds, rand);
                        //get the accuracy, here we used double because
                        //we need the accurate number ex. 92.21
                        acc = eval.pctCorrect();
                        System.out.println("SMO,"+acc);
                } catch (IOException e) {
                        e.printStackTrace();
                }
                catch (Exception e) {
                        e.printStackTrace();
                }
        }
}
```

Appendix C

SBIT Implementation

```
import java.io.FileNotFoundException;
import weka.classifiers.trees.RandomForest;
import weka.core.Instance;
import weka.core.Instances;
import weka.core.converters.ConverterUtils.DataSource;

public class SBIT {
  public static void main(String [] argv) throws FileNotFoundException {
    try {
      // the source dataset filenames
      String[] sourceDatasets = {"TBot.arff","Zeus.arff",
                "osx_trojan.arff","RBot.arff"};
      // the target dataset filename
      String targetDatasetName = "Sogou.arff";
      //load the target dataset
      DataSource target = new DataSource("botnet-data/"+targetDatasetName);
      Instances targetData = target.getDataSet();
      targetData.setClassIndex(targetData.numAttributes() - 1);

      //create a randomforest model using the target data
      RandomForest rf1 = new RandomForest();
      rf1.buildClassifier(targetData);

      //test dataset
      String testData = "ISCX_Testing_new_Sogou_TEST.arff";
      DataSource test = new DataSource("botnet-data/"+testData);
      Instances testDataset = test.getDataSet();
      testDataset.setClassIndex(testDataset.numAttributes() - 1);

      //create an empty dataset to copy instances to it temporarily
      //later we concatenate this with the target dataset
      Instances dataToTransfer = new Instances(testDataset,0,0);

      //loop through source datasets
      int n = sourceDatasets.length;
      for(int i = 0; i < n; i++){
        //load the ith source dataset
        DataSource source1 = new DataSource("botnet-data/"+sourceDatasets[i]);
        Instances sourceData1 = source1.getDataSet();
        sourceData1.setClassIndex(sourceData1.numAttributes() - 1);
        //loop through instances of ith source dataset
        for(int j = 0; j < sourceData1.numInstances(); j++){
          //get Instance object of current instance
          Instance srcInst = sourceData1.instance(j);
          //get attr values in a double array
          double[] srcAttrs = new double[srcInst.numAttributes()-1];
          for(int att = 0; att < (srcInst.numAttributes()-1); att++)
            srcAttrs[att] = srcInst.value(att);
          //loop through instances of target dataset
          for(int k = 0; k < targetData.numInstances(); k++){
            //get Instance object of current instance
```

```
            Instance trgInst = targetData.instance(k);
            //get attr values in a double array
            double[] trgAttrs = new double[trgInst.numAttributes()-1];
            for(int att = 0; att < (trgInst.numAttributes()-1); att++)
              trgAttrs[att] = trgInst.value(att);

            //now we have attr values for both src and trg instances
            //we can compute similarity between them
            double taniSim = Distance.tanimotoSimiarity(srcAttrs, trgAttrs);
            double ellenbergSim = Distance.ellenbergSimilarity(srcAttrs, trgAttrs);
            double gleasonSim = Distance.gleasonSimilarity(srcAttrs, trgAttrs);
            double ruzickaSim = Distance.ruzickaSimilarity(srcAttrs, trgAttrs);
            double brayCurtisSim = Distance.brayCurtisSimilarity(srcAttrs, trgAttrs);
            //add current source instance of it passes the similarity thresholds
            if(taniSim > 0.55 &&
                ellenbergSim > 0.55 &&
                gleasonSim > 0.55 &&
                ruzickaSim > 0.55 && brayCurtisSim > 0.55
                ){
              dataToTransfer.add(srcInst);
            }
          }
        }
      }


      // here we add the instances we have selected
      Instances newTargetData = new Instances(dataToTransfer);
      newTargetData.addAll(targetData);

      //create a randomforest model using the NEW target data
      //i.e. data after selecting instances from source datasets
      RandomForest rf2 = new RandomForest();
      rf2.buildClassifier(newTargetData);

      //keep track of class values for actuals and predicted
      String[] actuals = new String[testDataset.numInstances()];
      String[] rf1Predicted = new String[testDataset.numInstances()];
      String[] rf2Predicted = new String[testDataset.numInstances()];

      //Now loop through instances of test data and get
      //predictions for both RF models
      for (int i = 0; i < testDataset.numInstances(); i++) {
        //get Instance object of current instance
        Instance newInst = testDataset.instance(i);
        double actual = newInst.classValue();
        String actualClass = targetData.classAttribute().value((int) actual);
        actuals[i] = actualClass;

        //call classifyInstance, which returns a double value for the class
        //classify using model created using original target dataset
        double predicted = rf1.classifyInstance(newInst);
        String rf1PredictedClass = targetData.classAttribute().value((int) predicted);
        rf1Predicted[i] = rf1PredictedClass;
        //classify using model created using original target dataset
        predicted = rf2.classifyInstance(newInst);
        String rf2PredictedClass = targetData.classAttribute().value((int) predicted);
        rf2Predicted[i] = rf2PredictedClass;
      }

      System.out.println("=======================");
      double rf1Accuracy = compareResults(actuals, rf1Predicted);
      System.out.println("RF1 Accuracy:  " + rf1Accuracy );
      double rf2Accuracy = compareResults(actuals, rf2Predicted);
      System.out.println("RF2 (SBIT) Accuracy:  " + rf2Accuracy );

    } catch (Exception e) {
      // TODO Auto-generated catch block
      e.printStackTrace();
    }
  }
```

```
  /** a small function to compute accuracy
   *
   * @param actual the actual values
   * @param predicted the predicted values
   * @return accuracy
   */
  public static double compareResults(String actual[], String predicted[]){
    double equals = 0;
    //int unequals = 0;
    for(int i = 0; i < actual.length; i++){
      if(actual[i].equals(predicted[i])){
        equals++;
      }

    }
    return ((equals/actual.length)*100);
  }
}
```

Appendix D

CB-SBIT Implementation

```
import java.io.FileNotFoundException;
import weka.classifiers.trees.RandomForest;
//import weka.core.AttributeStats;
import weka.core.Instance;
import weka.core.Instances;
import weka.core.converters.ConverterUtils.DataSource;
import weka.filters.supervised.instance.SpreadSubsample;
import weka.filters.Filter;

public class ClassBalancedSBIT {

  public static void main(String [] argv) throws FileNotFoundException {
    try {
      // the source dataset filenames
      String[] sourceDatasets = {"TBot.arff","Zeus.arff",
                "osx_trojan.arff","RBot.arff"};
      // the target dataset filename
      String targetDatasetName = "Sogou.arff";
      //load the target dataset
      DataSource target = new DataSource("botnet-data/"+targetDatasetName);
      Instances targetData = target.getDataSet();
      targetData.setClassIndex(targetData.numAttributes() - 1);

      //create a randomforest model using the target data
      RandomForest rf1 = new RandomForest();
      rf1.buildClassifier(targetData);

      //test dataset
      String testData = "ISCX_Testing_new_Sogou_TEST.arff";
      DataSource test = new DataSource("botnet-data/"+testData);
      Instances testDataset = test.getDataSet();
      testDataset.setClassIndex(testDataset.numAttributes() - 1);

      //create an empty dataset to copy instances to it temporarily
      //later we concatenate this with the target dataset
      Instances dataToTransfer = new Instances(testDataset,0,0);

      //loop through source datasets
      int n = sourceDatasets.length;
      for(int i = 0; i < n; i++){
        //load the ith source dataset
        DataSource source1 = new DataSource(sourceDatasets[i]);
        Instances sourceData1 = source1.getDataSet();
        sourceData1.setClassIndex(sourceData1.numAttributes() - 1);
        //loop through instances of ith source dataset
        for(int j = 0; j < sourceData1.numInstances(); j++){
          //get Instance object of current instance
          Instance srcInst = sourceData1.instance(j);
          //get attr values in a double array
          double[] srcAttrs = new double[srcInst.numAttributes()-1];
          for(int att = 0; att < (srcInst.numAttributes()-1); att++)
```

```
        srcAttrs[att] = srcInst.value(att);
      //loop through instances of target dataset
      for(int k = 0; k < targetData.numInstances(); k++){
        //get Instance object of current instance
        Instance trgInst = targetData.instance(k);
        //get attr values in a double array
        double[] trgAttrs = new double[trgInst.numAttributes()-1];
        for(int att = 0; att < (trgInst.numAttributes()-1); att++)
          trgAttrs[att] = trgInst.value(att);

        //now we have attr values for both src and trg instances
        //we can compute similarity between them
        double taniSim  = Distance.tanimotoSimiarity(srcAttrs, trgAttrs);
        double ellenbergSim = Distance.ellenbergSimilarity(srcAttrs, trgAttrs);
        double gleasonSim = Distance.gleasonSimilarity(srcAttrs, trgAttrs);
        double ruzickaSim = Distance.ruzickaSimilarity(srcAttrs, trgAttrs);
        double brayCurtisSim = Distance.brayCurtisSimilarity(srcAttrs, trgAttrs);
        //add current source instance of it passes the similarity thresholds
        if(taniSim > 0.55 &&
            ellenbergSim > 0.55 &&
            gleasonSim > 0.55 &&
            ruzickaSim > 0.55 && brayCurtisSim > 0.55
            ){
          dataToTransfer.add(srcInst);
        }

      }
    }
  }
  //first we balance the data to transfer
  //resample majority class (down sample)
  SpreadSubsample ss = new SpreadSubsample();
  ss.setDistributionSpread(1.0);
  ss.setInputFormat(dataToTransfer);
  //here is where we apply the filter
  Instances balancedDataToTransfer = Filter.useFilter(dataToTransfer, ss);

  // here we add the instances we have selected
  Instances newTargetData = new Instances(balancedDataToTransfer);
  newTargetData.addAll(targetData);

  //uncomment the following lines to check the class distributions

  //AttributeStats stats = targetData.attributeStats(targetData.classIndex());
  //int[] nominalCounts = stats.nominalCounts;
  //System.out.println(Arrays.toString(nominalCounts));

  //stats = dataToTransfer.attributeStats(dataToTransfer.classIndex());
  //nominalCounts = stats.nominalCounts;
  //System.out.println(Arrays.toString(nominalCounts));

  //System.out.println("Original Target Dataset size: "+targetData.numInstances());
  //System.out.println("New Target Dataset size: "+newTargetData.numInstances());
  //System.out.println("dataToTransfer Dataset size: "+dataToTransfer.numInstances());

  //create a randomforest model using the NEW target data
  //i.e. data after selecting instances from source datasets
  RandomForest rf2 = new RandomForest();
  rf2.buildClassifier(newTargetData);

  //keep track of class values for actuals and predicted
  String[] actuals = new String[testDataset.numInstances()];
  String[] rf1Predicted = new String[testDataset.numInstances()];
  String[] rf2Predicted = new String[testDataset.numInstances()];

  //Now loop through instances of test data and get
  //predictions for both RF models
  for (int i = 0; i < testDataset.numInstances(); i++) {
    //get Instance object of current instance
    Instance newInst = testDataset.instance(i);
    double actual = newInst.classValue();
```

```
        String actualClass = targetData.classAttribute().value((int) actual);
        actuals[i] = actualClass;

        //call classifyInstance, which returns a double value for the class
        //classify using model created using original target dataset
        double predicted = rf1.classifyInstance(newInst);
        String rf1PredictedClass = targetData.classAttribute().value((int) predicted);
        rf1Predicted[i] = rf1PredictedClass;
        //classify using model created using original target dataset
        predicted = rf2.classifyInstance(newInst);
        String rf2PredictedClass = targetData.classAttribute().value((int) predicted);
        rf2Predicted[i] = rf2PredictedClass;
      }

      System.out.println("======================");
      double rf1Accuracy = compareResults(actuals, rf1Predicted);
      System.out.println("RF1 Accuracy:  " + rf1Accuracy );
      double rf2Accuracy = compareResults(actuals, rf2Predicted);
      System.out.println("RF2 (CB-SBIT) Accuracy:  " + rf2Accuracy );

    } catch (Exception e) {
      e.printStackTrace();
    }
  }

  /** a small function to compute accuracy
   *
   * @param actual the actual values
   * @param predicted the predicted values
   * @return accuracy
   */
  public static double compareResults(String actual[], String predicted[]){
    double equals = 0;
    //int unequals = 0;
    for(int i = 0; i < actual.length; i++){
      if(actual[i].equals(predicted[i])){
        equals++;
      }
    }
    return ((equals/actual.length)*100);
  }
}
```

Bibliography

Abdul Kadir, A. F., Stakhanova, N. and Ghorbani, A. A. (2015), Android botnets: What urls are telling us, *in* M. Qiu, S. Xu, M. Yung and H. Zhang, eds, 'Network and System Security', Springer International Publishing, Cham, pp. 78–91.

Abdulla, S., Ramadass, S. and Altyeb, A. A. (2014), 'kenfis: knn-based evolving neuro-fuzzy inference system for computer worms detection', *Journal of Intelligent and Fuzzy Systems* **26**, 1893–1908.

Abuadlla, Y., Kvascev, G., Gajin, S. and Jovanović, Z. (2014), 'Flow-based anomaly intrusion detection system using two neural network stages', *Computer Science and Information Systems* **11**(2), 601–622.

Acarali, D., Rajarajan, M., Komninos, N. and Herwono, I. (2016), 'Survey of approaches and features for the identification of http-based botnet traffic', *Journal of Network and Computer Applications* **76**, 1 – 15.

URL: *http://www.sciencedirect.com/science/article/pii/S1084804516302363*

Agency, T. N. C. (2017), 'Uk internet users potential victims of serious cyber attack'.

URL: *http://www.nationalcrimeagency.gov.uk/news/723-uk-internet-users-potential-victims-of-serio*

Aggarwal, C. C. (2013), *Outlier Analysis*, Springer Publishing Company, Incorporated.

ALLISON, P. D. (2000), 'Multiple imputation for missing data: A cautionary tale', *Sociological Methods & Research* **28**(3), 301–309.

URL: *https://doi.org/10.1177/0049124100028003003*

Alothman, B. (2018*a*), 'Raw network traffic data preprocessing and preparation for automatic analysis', *International Conference On Cyber Incident Response, Coordination, Containment & Control (Cyber Incident) - 2018* .

Alothman, B. (2018*b*), 'Similarity based instance transfer learning for botnet detection', *International Journal of Intelligent Computing Research (IJICR)* **9**, 880—-889.

Alothman, B., Janicke, H. and Yerima, S. Y. (2018), Class balanced similarity-based instance transfer learning for botnet family classification, *in* L. Soldatova, J. Vanschoren, G. Papadopoulos and M. Ceci, eds, 'Discovery Science', Springer International Publishing, Cham, pp. 99–113.

Alothman, B. and Rattadilok, P. (2017), Android botnet detection: An integrated source code mining approach, *in* '2017 12th International Conference for Internet Technology and Secured Transactions (ICITST)', pp. 111–115.

Alzaylaee, M. K., Yerima, S. Y. and Sezer, S. (2016), 'Dynalog: An automated dynamic analysis framework for characterizing android applications', *CoRR* **abs/1607.08166**.

Alzaylaee, M. K., Yerima, S. Y. and Sezer, S. (2017), 'Improving dynamic analysis of android apps using hybrid test input generation', *CoRR* **abs/1705.06691**.

Argyriou, A., Evgeniou, T. and Pontil, M. (2008), 'Convex multi-task feature learning', *Mach. Learn.* **73**(3), 243–272.

URL: *http://dx.doi.org/10.1007/s10994-007-5040-8*

Argyriou, A., Micchelli, C. A., Pontil, M. and Ying, Y. (2007), A spectral regularization framework for multi-task structure learning, *in* 'Proceedings of the 20th International Conference on Neural Information Processing Systems', NIPS'07, Curran Associates Inc., USA, pp. 25–32.

URL: *http://dl.acm.org/citation.cfm?id=2981562.2981566*

Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D. and McDaniel, P. (2014), Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps, *in* 'Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation', PLDI '14, ACM, New York, NY, USA, pp. 259–269.

URL: *http://doi.acm.org/10.1145/2594291.2594299*

Bai, G., Wu, Y., Sun, J., Wu, J., Liu, Y., Zhang, Q. and Dong, J. S. (2016), 'Droidpf: a framework for automatic verification of android applications'.

Bartos, K., Sofka, M. and Franc, V. (2016), Optimized invariant representation of network traffic for detecting unseen malware variants, *in* '25th USENIX Security Symposium (USENIX Security 16)', USENIX Association, Austin, TX, pp. 807–822.

URL: *https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/bartos*

Beghdad, R. (2008), 'Critical study of neural networks in detecting intrusions', *Comput. Secur.* **27**(5-6), 168–175.

URL: *http://dx.doi.org/10.1016/j.cose.2008.06.001*

Benjamin, V. and Chen, H. (2013), Machine learning for attack vector identification in malicious source code, *in* 'IEEE ISI 2013 - 2013 IEEE International Conference on Intelligence and Security Informatics: Big Data, Emergent Threats, and Decision-Making in Security Informatics', pp. 21–23.

Bijalwan, A., Chand, N., Pilli, E. S. and Krishna, C. R. (2016), 'Botnet analysis using ensemble classifier', *Perspectives in Science* **8**, 502 – 504. Recent Trends in Engineering and Material Sciences.

URL: *http://www.sciencedirect.com/science/article/pii/S2213020916301422*

Binsalleeh, H., Ormerod, T., Boukhtouta, A., Sinha, P., Youssef, A. M., Debbabi, M. and Wang, L. (2010), 'On the analysis of the zeus botnet crimeware toolkit', `http://www.ncfta.ca/papers/On_the_Analysis_of_the_ZeuS_Botnet_Crimeware.pdfwww.ncfta.ca`.

Bishop, C. M. (2006), *Pattern Recognition and Machine Learning (Information Science and Statistics)*, Springer-Verlag, Berlin, Heidelberg.

Borgaonkar, R. (2010), An analysis of the asprox botnet, *in* '2010 Fourth International Conference on Emerging Security Information, Systems and Technologies', pp. 148–153.

Boulesteix, A.-L. and Strimmer, K. (2007), 'Partial least squares: a versatile tool for the analysis of high-dimensional genomic data', *Brief Bioinform* **8**(1), 32–44.

Bradley, A. P. (1997), 'The use of the area under the roc curve in the evaluation of machine learning algorithms', *Pattern Recogn.* **30**(7), 1145–1159.

URL: *http://dx.doi.org/10.1016/S0031-3203(96)00142-2*

Bramer, M. (2013), *Principles of Data Mining*, 2nd edn, Springer Publishing Company, Incorporated.

Breunig, M., Kriegel, H.-P., Ng, R. T. and Sander, J. (2000), Lof: Identifying density-based local outliers, *in* 'PROCEEDINGS OF THE 2000 ACM SIGMOD

INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA', ACM, pp. 93–104.

Chawla, N. (2005), *Data Mining for Imbalanced Datasets: An Overview*, Vol. 5, Springer.

Chawla, N. V. (2010), *Data Mining for Imbalanced Datasets: An Overview*, Springer US, Boston, MA, pp. 875–886.

URL: *https://doi.org/10.1007/978-0-387-09823-4_45*

Chawla, N. V., Bowyer, K. W., Hall, L. O. and Kegelmeyer, P. W. (2002), 'Smote: Synthetic minority over-sampling technique', *J. Artif. Int. Res.* **16**(1), 321–357.

URL: *http://dl.acm.org/citation.cfm?id=1622407.1622416*

CodeAnalyzer (2017), 'Codeanalyzer'. Accessed 29 Nov 2017.

URL: *http://www.codeanalyzer.teel.ws*

Cooke, E., Jahanian, F. and McPherson, D. (2005), The zombie roundup: Understanding, detecting, and disrupting botnets, *in* 'Proceedings of the Steps to Reducing Unwanted Traffic on the Internet on Steps to Reducing Unwanted Traffic on the Internet Workshop', SRUTI'05, USENIX Association, Berkeley, CA, USA, pp. 6–6.

URL: *http://dl.acm.org/citation.cfm?id=1251282.1251288*

Costa, K. A., Pereira, L. A., Nakamura, R. Y., Pereira, C. R., Papa, J. P. and Falcão, A. X. (2015), 'A nature-inspired approach to speed up optimum-path forest clustering and its application to intrusion detection in computer networks', *Information Sciences* **294**, 95 – 108. Innovative Applications of Artificial Neural Networks in Engineering.

URL: *http://www.sciencedirect.com/science/article/pii/S0020025514009311*

Crosbie, J. (2016), 'The internet of things helped a ddos attack destroy the internet', `https://www.inverse.com/article/22591-internet-of-things-ddos-attack`.

Dai, W., Yang, Q., Xue, G.-R. and Yu, Y. (2007), Boosting for transfer learning, *in* 'Proceedings of the 24th International Conference on Machine Learning', ICML '07, ACM, New York, NY, USA, pp. 193–200.
URL: *http://doi.acm.org/10.1145/1273496.1273521*

Davis, J. J. and Clark, A. J. (2011), 'Data preprocessing for anomaly based network intrusion detection: A review', *Comput. Secur.* **30**(6-7), 353–375.
URL: *http://dx.doi.org/10.1016/j.cose.2011.05.008*

Demarest, J. (2014), 'Taking down botnets: Statement before the senate judiciary committee, subcommittee on crime and terrorism', `https://www.fbi.gov/news/testimony/taking-down-botnets`.

Deza, M. M. and Deza, E. (2009), *Encyclopedia of Distances*, Springer Berlin Heidelberg.

Dietrich, C. J., Rossow, C., Freiling, F. C., Bos, H., van Steen, M. and Pohlmann, N. (2011), 'On botnets that use dns for command and control', `http://www.cj2s.de/On-Botnets-that-use-DNS-for-Command-and-Control.pdf`.

Draper-Gil, G., Lashkari, A. H., Mamun, M. S. I. and Ghorbani, A. A. (2016), Characterization of encrypted and vpn traffic using time-related features, *in* 'ICISSP'.

Dua, S. and Du, X. (2011), *Data Mining and Machine Learning in Cybersecurity*, 1st edn, Auerbach Publications, Boston, MA, USA.

Eaton, E. and desJardins, M. (2011), Selective transfer between learning tasks using task-based boosting, *in* 'Proceedings of the 25th AAAI Conference on Artificial Intelligence (AAAI-11)', AAAI Press, pp. 337–342.

Evgeniou, T. and Pontil, M. (2004), Regularized multi–task learning, *in* 'Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining', KDD '04, ACM, New York, NY, USA, pp. 109–117.
URL: *http://doi.acm.org/10.1145/1014052.1014067*

Feldman, R. and Sanger, J. (2006), *Text Mining Handbook: Advanced Approaches in Analyzing Unstructured Data*, Cambridge University Press, New York, NY, USA.

Feng, L., Wang, H., Han, Q., Zhao, Q. and Song, L. (2014), Modeling peer-to-peer botnet on scale-free network, *in* 'Abstract and Applied Analysis'. http://dx.doi.org/10.1155/2014/212478.

Fowler, C. A. and Hammel, R. J. (2014), Converting pcaps into weka mineable data, *in* '2014 15th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)', Vol. 00, pp. 1–6.
URL: *doi.ieeecomputersociety.org/10.1109/SNPD.2014.6888681*

Garcia, S. and Pechoucek, M. (2016), Detecting the behavioral relationships of malware connections, *in* 'Proceedings of the 1st International Workshop on AI for Privacy and Security', PrAISe '16, ACM, New York, NY, USA, pp. 8:1–8:5.
URL: *http://doi.acm.org/10.1145/2970030.2970038*

Ghafir, I., Prenosil, V., Hammoudeh, M., Baker, T., Jabbar, S., Khalid, S. and Jaf, S. (2018), 'Botdet: A system for real time botnet command and control traffic detection', *IEEE Access* **6**, 38947–38958.

Gonzalez, H., Stakhanova, N. and Ghorbani, A. A. (2015), Droidkin: Lightweight detection of android apps similarity, *in* J. Tian, J. Jing and M. Srivatsa, eds,

'International Conference on Security and Privacy in Communication Networks', Springer International Publishing, Cham, pp. 436–453.

Gordon, M. I., Kim, D., Perkins, J., Gilham, L., Nguyen, N. and Rinard, M. (2015), Information-flow analysis of Android applications in DroidSafe, *in* 'Proceedings of the 22nd Annual Network and Distributed System Security Symposium (NDSS)'.

Gu, G., Perdisci, R., Zhang, J. and Lee, W. (2008), Botminer: Clustering analysis of network traffic for protocol- and structure-independent botnet detection, *in* 'Proceedings of the 17th Conference on Security Symposium', SS'08, USENIX Association, Berkeley, CA, USA, pp. 139–154.

URL: *http://dl.acm.org/citation.cfm?id=1496711.1496721*

Gu, G., Porras, P., Yegneswaran, V., Fong, M. and Lee, W. (2007), Bothunter: Detecting malware infection through ids-driven dialog correlation, *in* 'Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium', SS'07, USENIX Association, Berkeley, CA, USA, pp. 12:1–12:16.

URL: *http://dl.acm.org/citation.cfm?id=1362903.1362915*

Guo, C. and Berkhahn, F. (2016), 'Entity embeddings of categorical variables', *CoRR* **abs/1604.06737**.

URL: *http://arxiv.org/abs/1604.06737*

Haddadi, F., Runkel, D., Zincir-Heywood, A. N. and Heywood, M. I. (2014), On botnet behaviour analysis using gp and c4.5, *in* 'Proceedings of the Companion Publication of the 2014 Annual Conference on Genetic and Evolutionary Computation', GECCO Comp '14, ACM, New York, NY, USA, pp. 1253–1260.

URL: *http://doi.acm.org/10.1145/2598394.2605435*

Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P. and Witten, I. H. (2009), 'The weka data mining software: An update', *SIGKDD Explor. Newsl.* **11**(1), 10–18.
URL: *http://doi.acm.org/10.1145/1656274.1656278*

Haykin, S. (2007), *Neural Networks: A Comprehensive Foundation (3rd Edition)*, Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

He, H. and Ma, Y. (2013), *Imbalanced Learning: Foundations, Algorithms, and Applications*, 1st edn, Wiley-IEEE Press.

Healey, J. and Knake, R. K. (2018), 'Zero botnets: Building a global effort to clean up the internet', `https://cfrd8-files.cfr.org/sites/default/files/report_pdf/CSR83_HealeyKnake_Botnets_0.pdf`.

Hodge, V. and Austin, J. (2004), 'A survey of outlier detection methodologies', *Artif. Intell. Rev.* **22**(2), 85–126.
URL: *https://doi.org/10.1023/B:AIRE.0000045502.10941.a9*

International Telecommunication Union (2017), 'Global cybersecurity index (gci) 2017'. `https://www.itu.int/dms_pub/itu-d/opb/str/d-str-gci.01-2017-pdf-e.pdf`.

Jadidi, Z., Muthukkumarasamy, V. and Sithirasenan, E. (2013), Metaheuristic algorithms based flow anomaly detector, *in* '2013 19th Asia-Pacific Conference on Communications (APCC)', pp. 717–722.

Japkowicz, N. and Shah, M. (2011), *Evaluating Learning Algorithms: A Classification Perspective*, Cambridge University Press, New York, NY, USA.

Jolliffe, I. (1986), *Principal Component Analysis*, Springer Verlag.

Joorabchi, M. E. and Mesbah, A. (2012), Reverse engineering ios mobile applications, *in* '2012 19th Working Conference on Reverse Engineering', pp. 177–186.

Junaid, M., Liu, D. and Kung, D. (2016), 'Dexteroid: Detecting malicious behaviors in android apps using reverse-engineered life cycle models', *Computers & Security* **59**, 92 – 117.

URL: *http://www.sciencedirect.com/science/article/pii/S0167404816300037*

Kabakus, A. T. and Dogru, I. A. (2018), 'An in-depth analysis of android malware using hybrid techniques', *Digital Investigation* **24**, 25 – 33.

URL: *http://www.sciencedirect.com/science/article/pii/S1742287617303183*

Kalita, E. (2017), *WannaCry Ransomware Attack: Protect Yourself from WannaCry Ransomware Cyber Risk and Cyber War*, Independently published.

Kang, B., Yerima, S. Y., McLaughlin, K. and Sezer, S. (2016), N-opcode analysis for android malware classification and categorization, *in* 'Cyber Security And Protection Of Digital Services', IEEE, pp. 1–7.

Kharouni, L. (2009), 'Sdbot irc botnet continues to make waves', `http://www.trendmicro.com/cloud-content/us/pdfs/security-intelligence/white-papers/wp_sdbot_irc_botnet_continues_to_make_waves_pub.pdf`.

Kirubavathi, G. and Anitha, R. (2016), 'Botnet detection via mining of traffic flow characteristics', *Computers & Electrical Engineering* **50**, 91 – 101.

URL: *http://www.sciencedirect.com/science/article/pii/S0045790616000148*

Koschke, R. (2005), What architects should know about reverse engineering and rengineering, *in* 'WICSA', IEEE Computer Society, pp. 4–10.

Kuhn, M. and Johnson, K. (2013), *Applied Predictive Modeling*, Springer, New York, Heidelberg, Dordrecht, London.

URL: *https://dl.dropboxusercontent.com/u/108263707/_book/KuhnJohnson2013apm.pdf*

Lang, K. (2007), '20 newsgroups data set', *MIT* .

URL: *http://www.ai.mit.edu/people/jrennie/20Newsgroups/*

Larose, D. T. (2004), *Discovering Knowledge in Data: An Introduction to Data Mining*, Wiley-Interscience.

Lawrence, N. D. and Platt, J. C. (2004), Learning to learn with the informative vector machine, *in* 'Proceedings of the Twenty-first International Conference on Machine Learning', ICML '04, ACM, New York, NY, USA, pp. 65–.

URL: *http://doi.acm.org/10.1145/1015330.1015382*

Lee, S.-I., Chatalbashev, V., Vickrey, D. and Koller, D. (2007), Learning a meta-level prior for feature relevance from multiple related tasks, *in* 'Proceedings of the 24th International Conference on Machine Learning', ICML '07, ACM, New York, NY, USA, pp. 489–496.

URL: *http://doi.acm.org/10.1145/1273496.1273558*

Liao, H.-J., Lin, C.-H. R., Lin, Y.-C. and Tung, K.-Y. (2013), 'Intrusion detection system: A comprehensive review', *Journal of Network and Computer Applications* **36**(1), 16 – 24.

URL: *http://www.sciencedirect.com/science/article/pii/S1084804512001944*

Liu, B., Xiao, Y. and Hao, Z. (2018), 'A selective multiple instance transfer learning method for text categorization problems', *Knowledge-Based Systems* **141**, 178 – 187.

URL: *http://www.sciencedirect.com/science/article/pii/S0950705117305415*

Liu, H. and Motoda, H. (1998), *Feature Selection for Knowledge Discovery and Data Mining*, Kluwer Academic Publishers, Norwell, MA, USA.

Liu, H. and Motoda, H. (2007), *Computational Methods of Feature Selection (Chapman & Hall/Crc Data Mining and Knowledge Discovery Series)*, Chapman & Hall/CRC.

McLaughlin, N., Martinez del Rincon, J., Kang, B., Yerima, S., Miller, P., Sezer, S., Safaei, Y., Trickel, E., Zhao, Z., Doupé, A. and Joon Ahn, G. (2017), Deep android malware detection, *in* 'Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy', CODASPY '17, ACM, New York, NY, USA, pp. 301–308.

URL: *http://doi.acm.org/10.1145/3029806.3029823*

Middleton, P., Kjeldsen, P. and Tully, J. (2013), 'Forecast: The internet of things, worldwide', `https://www.gartner.com/doc/2625419/forecast-internet-things-worldwide-`.

Mihalkova, L., Huynh, T. and Mooney, R. J. (2007), Mapping and revising markov logic networks for transfer learning, *in* 'Proceedings of the 22Nd National Conference on Artificial Intelligence - Volume 1', AAAI'07, AAAI Press, pp. 608–614.

URL: *http://dl.acm.org/citation.cfm?id=1619645.1619743*

Mirjalili, S. and Hashim, S. Z. M. (2010), A new hybrid psogsa algorithm for function optimization, *in* '2010 International Conference on Computer and Information Application', pp. 374–377.

Müller, H. A., Jahnke, J. H., Smith, D. B., Storey, M.-A., Tilley, S. R. and Wong, K. (2000), Reverse engineering: A roadmap, *in* 'Proceedings of the Conference on The Future of Software Engineering', ICSE '00, ACM, New York, NY, USA, pp. 47–60.

URL: *http://doi.acm.org/10.1145/336512.336526*

Nikola, M., Dehghantanha, A. and Raymond, C. K.-K. (2017), 'Machine learning aided android malware classification', *Computers & Electrical Engineering* **61**, 266–274.
URL: *http://usir.salford.ac.uk/41554/*

Orebaugh, A., Ramirez, G., Beale, J. and Wright, J. (2007), *Wireshark & Ethereal Network Protocol Analyzer Toolkit*, Syngress Publishing.

Pan, S. J. and Yang, Q. (2010), 'A survey on transfer learning', *IEEE Trans. on Knowl. and Data Eng.* **22**(10), 1345–1359.
URL: *http://dx.doi.org/10.1109/TKDE.2009.191*

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M. and Duchesnay, E. (2011), 'Scikit-learn: Machine learning in Python', *Journal of Machine Learning Research* **12**, 2825–2830.

Pigott, T. D. (2001), 'A review of methods for missing data', *Educational Research and Evaluation* **7**(4), 353–383.
URL: *http://www.tandfonline.com/doi/abs/10.1076/edre.7.4.353.8937*

Platt, J. C. (1998), Sequential minimal optimization: A fast algorithm for training support vector machines, Technical report, ADVANCES IN KERNEL METHODS - SUPPORT VECTOR LEARNING.

Quinlan, J. R. (1993), *C4.5: Programs for Machine Learning*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Rahman, G. and Islam, Z. (2011), A decision tree-based missing value imputation technique for data pre-processing, *in* 'Proceedings of the Ninth Australasian Data Mining Conference - Volume 121', AusDM '11, Australian Computer Society, Inc.,

Darlinghurst, Australia, Australia, pp. 41–50.

URL: *http://dl.acm.org/citation.cfm?id=2483628.2483635*

Rai, K., Syamala, M., Professor, D. and Guleria, A. (2016), 'Decision tree based algorithm for intrusion detection', *International Journal of Advanced Networking and Applications* **07**, 2828–2834.

Raina, R., Battle, A., Lee, H., Packer, B. and Ng, A. Y. (2007), Self-taught learning: Transfer learning from unlabeled data, *in* 'Proceedings of the 24th International Conference on Machine Learning', ICML '07, ACM, New York, NY, USA, pp. 759–766.

URL: *http://doi.acm.org/10.1145/1273496.1273592*

Rajabioun, R. (2011), 'Cuckoo optimization algorithm', *Applied Soft Computing* **11**(8), 5508 – 5518.

URL: *http://www.sciencedirect.com/science/article/pii/S1568494611001670*

Richardson, M. and Domingos, P. (2006), 'Markov logic networks', *Mach. Learn.* **62**(1-2), 107–136.

URL: *http://dx.doi.org/10.1007/s10994-006-5833-1*

Rish, I. (2001), An empirical study of the naive bayes classifier, *in* 'IJCAI 2001 workshop on empirical methods in artificial intelligence', Vol. 3, IBM New York, pp. 41–46.

Robinson, N. and Martin, K. (2017), 'Distributed denial of government: the estonian data embassy initiative', *Network Security* **2017**(9), 13 – 16.

URL: *http://www.sciencedirect.com/science/article/pii/S1353485817301149*

Rokach, L. and Maimon, O. (2014), *Data Mining With Decision Trees: Theory and Applications*, 2nd edn, World Scientific Publishing Co., Inc., River Edge, NJ, USA.

Samani, E. B. B., Jazi, H. H., Stakhanova, N. and Ghorbani, A. A. (2014), 'Towards effective feature selection in machine learning-based botnet detection approaches', *2014 IEEE Conference on Communications and Network Security* pp. 247–255.

Santafe, G., Inza, I. n. and Lozano, J. A. (2015), 'Dealing with the evaluation of supervised classification algorithms', *Artif. Intell. Rev.* **44**(4), 467–508.
URL: *http://dx.doi.org/10.1007/s10462-015-9433-y*

Schiller, C. and Binkley, J. (2007), *Botnets: The Killer Web Applications*, Syngress Publishing.

Sheen, S., Anitha, R. and Natarajan, V. (2015), 'Android based malware detection using a multifeature collaborative decision fusion approach', *Neurocomputing* **151**, 905 – 912.
URL: *http://www.sciencedirect.com/science/article/pii/S0925231214012739*

Silva, S. S., Silva, R. M., Pinto, R. C. and Salles, R. M. (2013), 'Botnets: A survey', *Computer Networks* **57**(2), 378 – 403. Botnet Activity: Analysis, Detection and Shutdown.
URL: *http://www.sciencedirect.com/science/article/pii/S1389128612003568*

Simonson, K. (2013), 'Nate silver, the signal and the noise: Why so many predictions fail—but some don't', *Business Economics* **48**(1), 82–84.
URL: *https://EconPapers.repec.org/RePEc:pal:buseco:v:48:y:2013:i:1:p:82-84*

Sood, A. K., Zeadally, S. and Enbody, R. J. (2016), 'An empirical study of http-based financial botnets', *IEEE Transactions on Dependable and Secure Computing* **13**(2), 236–251.

Steinwart, I. and Christmann, A. (2008), *Support Vector Machines*, 1st edn, Springer-Verlag New York.

Stevanovic, M. and Pedersen, J. (2013), Machine learning for identifying botnet network traffic.

Stevanovic, M. and Pedersen, J. M. (2014), An efficient flow-based botnet detection using supervised machine learning, *in* '2014 International Conference on Computing, Networking and Communications (ICNC)', pp. 797–801.

Stiborek, J., Pevný, T. and Rehák, M. (2018), 'Multiple instance learning for malware classification', *Expert Systems with Applications* **93**, 346 – 357.
URL: *http://www.sciencedirect.com/science/article/pii/S0957417417307170*

Stover, S., Dittrich, D., Hernandez, J. and Dietrich, S. (2007), Analysis of the storm and nugache trojans: P2P is here, *in* ';login'.

Suarez-Tangil, G., Tapiador, J. E., Peris-Lopez, P. and Blasco, J. (2014), 'Dendroid: A text mining approach to analyzing and classifying code structures in android malware families', *Expert Systems with Applications* **41**(4, Part 1), 1104 – 1117.
URL: *http://www.sciencedirect.com/science/article/pii/S0957417413006088*

Sun, G., Liang, L., Chen, T., Xiao, F. and Lang, F. (2018), 'Network traffic classification based on transfer learning', *Computers & Electrical Engineering* .
URL: *http://www.sciencedirect.com/science/article/pii/S004579061732829X*

Team, D. (2016), 'dex2jar'. Accessed 22 Oct 2017.
URL: *https://sourceforge.net/projects/dex2jar/*

Team, J.-D. (2015), 'java-decompiler/jd-gui'. Accessed on 03-05-2018.
URL: *https://github.com/java-decompiler/jd-gui*

The Council of Economic Advisers (2018), 'The cost of malicious cyber activity to the u.s. economy'. `https://www.whitehouse.gov/wp-content/uploads/2018/02/The-Cost-of-Malicious-Cyber-Activity-to-the-U.S.-Economy.pdf`.

Tiirmaa-Klaar, H., Gassen, J., Gerhards-Padilla, E. and Martini, P. (2013), *Botnets*, Springer Publishing Company, Incorporated.

Torrey, L. and Shavlik, J. (2009), 'Transfer learning', *Handbook of Research on Machine Learning Applications. IGI Global* **3**, 17–35.

Tran, Q. A., Jiang, F. and Hu, J. (2012), A real-time netflow-based intrusion detection system with improved bbnn and high-frequency field programmable gate arrays, *in* '2012 IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications', pp. 201–208.

van den Berg, R. A., Hoefsloot, H. C., Westerhuis, J. A., Smilde, A. K. and van der Werf, M. J. (2006), 'Centering, scaling, and transformations: improving the biological information content of metabolomics data', *BMC Genomics* **7**(1), 142.

Verisign DDoS Report (2018), 'Q2 2018 ddos trends report: 52 percent of attacks employed multiple attack types', `https://blog.verisign.com/security/ddos-protection/q2-2018-ddos-trends-report-52-percent-of-attacks-employed-multiple-attack-types/`.

Wang, A., Chang, W., Chen, S. and Mohaisen, A. (2018), 'Delving into internet ddos attacks by botnets: Characterization and analysis', *IEEE/ACM Trans. Netw.* **26**(6), 2843–2855.
URL: *https://doi.org/10.1109/TNET.2018.2874896*

Wang, P., Sparks, S. and Zou, C. C. (2007), An advanced hybrid peer-to-peer botnet, *in* 'Proceedings of the First Conference on First Workshop on Hot Topics in Understanding Botnets', HotBots'07, USENIX Association, Berkeley, CA, USA, pp. 2–2.

URL: *http://dl.acm.org/citation.cfm?id=1323128.1323130*

Warrens, M. J. (2016), 'Inequalities between similarities for numerical data', *Journal of Classification* **33**(1), 141–148.

URL: *https://doi.org/10.1007/s00357-016-9200-z*

Wei, M., Gong, X. and Wang, W. (2015), Claim what you need: A text-mining approach on android permission request authorization, *in* '2015 IEEE Global Communications Conference (GLOBECOM)', pp. 1–6.

Weiss, S., Indurkhya, N., Zhang, T. and Damerau, F. (2004), *Text Mining: Predictive Methods for Analyzing Unstructured Information*, SpringerVerlag.

Weiss, S. M., Indurkhya, N. and Zhang, T. (2004), *Text Mining. Predictive Methods for Analyzing Unstructured Information*, 1 edn, Springer, Berlin.

Winter, P., Hermann, E. and Zeilinger, M. (2011), Inductive intrusion detection in flow-based network data using one-class support vector machines, *in* '2011 4th IFIP International Conference on New Technologies, Mobility and Security', pp. 1–5.

Witte, R., Li, Q., Zhang, Y. and Rilling, J. (2008), 'Text mining and software engineering: an integrated source code and document analysis approach', *IET Software* **2**(1), 3–16.

Wu, X., Kumar, V., Ross Quinlan, J., Ghosh, J., Yang, Q., Motoda, H., McLachlan, G. J., Ng, A., Liu, B., Yu, P. S., Zhou, Z.-H., Steinbach, M., Hand, D. J. and Steinberg,

D. (2007), 'Top 10 algorithms in data mining', *Knowl. Inf. Syst.* **14**(1), 1–37.

URL: *http://dx.doi.org/10.1007/s10115-007-0114-2*

Wu, X., Zhu, X., Wu, G.-Q. and Ding, W. (2014), 'Data mining with big data', *IEEE Trans. on Knowl. and Data Eng.* **26**(1), 97–107.

URL: *https://doi.org/10.1109/TKDE.2013.109*

Xue, F. and Qu, A. (2017), 'Variable Selection for Highly Correlated Predictors', *ArXiv e-prints* .

Yan, L. K. and Yin, H. (2012), Droidscope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis, *in* 'Proceedings of the 21st USENIX Conference on Security Symposium', Security'12, USENIX Association, Berkeley, CA, USA, pp. 29–29.

URL: *http://dl.acm.org/citation.cfm?id=2362793.2362822*

Yang, W., Zhang, Y., Li, J., Shu, J., Li, B., Hu, W. and Gu, D. (2015), Appspear: Bytecode decrypting and dex reassembling for packed android malware, *in* 'Proceedings of the 18th International Symposium on Research in Attacks, Intrusions, and Defenses - Volume 9404', RAID 2015, Springer-Verlag New York, Inc., New York, NY, USA, pp. 359–381.

URL: *http://dx.doi.org/10.1007/978-3-319-26362-5_17*

Yang, Z. and Yang, M. (2012), Leakminer: Detect information leakage on android with static taint analysis, *in* 'Proceedings of the 2012 Third World Congress on Software Engineering', WCSE '12, IEEE Computer Society, Washington, DC, USA, pp. 101–104.

URL: *http://dx.doi.org/10.1109/WCSE.2012.26*

Yang, Z., Yang, M., Zhang, Y., Gu, G., Ning, P. and Wang, X. S. (2013), Appintent: Analyzing sensitive data transmission in android for privacy leakage detection, *in* 'Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security', CCS '13, ACM, New York, NY, USA, pp. 1043–1054.
URL: *http://doi.acm.org/10.1145/2508859.2516676*

Yerima, S. Y., Sezer, S., McWilliams, G. and Muttik, I. (2013), A new android malware detection approach using bayesian classification, *in* 'Proceedings of the 2013 IEEE 27th International Conference on Advanced Information Networking and Applications', AINA '13, IEEE Computer Society, Washington, DC, USA, pp. 121–128.
URL: *http://dx.doi.org/10.1109/AINA.2013.88*

Yuan, R., Li, Z., Guan, X. and Xu, L. (2010), 'An svm-based machine learning method for accurate internet traffic classification', *Information Systems Frontiers* **12**(2), 149–156.

Zhang, J., Zulkernine, M. and Haque, A. (2008), 'Random-forests-based network intrusion detection systems', *Trans. Sys. Man Cyber Part C* **38**(5), 649–659.
URL: *http://dx.doi.org/10.1109/TSMCC.2008.923876*

Zhang, X., Breitinger, F. and Baggili, I. (2016), 'Rapid android parser for investigating dex files (rapid)', *Digital Investigation* **17**, 28 – 39.
URL: *http://www.sciencedirect.com/science/article/pii/S1742287616300305*

Zhao, D., Traore, I., Sayed, B., Lu, W., Saad, S., Ghorbani, A. and Garant, D. (2013), 'Botnet detection based on traffic behavior analysis and flow intervals', *Computers & Security* **39**, 2 – 16. 27th IFIP International Information Security Conference.
URL: *http://www.sciencedirect.com/science/article/pii/S0167404813000837*

Zhao, J., Shetty, S. and Pan, J. W. (2017), Feature-based transfer learning for network security, *in* 'MILCOM 2017 - 2017 IEEE Military Communications Conference (MILCOM)'.